University of Zurich[UZH]

Department of Informatics

MSc Thesis

Application of Higher–Order Network Models to Representation Learning in Sequential Data

Michael Markus Studer
of Maschwanden ZH, Switzerland

Student–ID: 92-911-544
mstuder72@gmail.com

November 18, 2020

supervised by Prof. Dr. Ingo Scholtes

Application of Higher-Order Network Models to Representation Learning in Sequential Data

Master Thesis Proposal Michael Studer (92-911-544) May 18th–November 11th, 2020
Data Analytics Group, University of Zürich
Supervised by Prof. Dr. Ingo Scholtes

Background

Unsupervised methods to learn low-dimensional representations of complex data are important tools to address the problem of feature extraction in machine learning [1]. Applying this concept to machine learning in graphs, researchers have recently developed graph representation learning or network embedded techniques[4, 8], which have been successfully applied to problems such as link prediction, node classification, clustering, or network visualization[5]. The common of those techniques is to learn a mapping function that maps the nodes of a graph to points in a low-dimensional vector space such that geometric distances between nodes reflect the topology of the network. Existing algorithms that approach this problem commonly utilize an edge-centric view that focuses on the **topology of edges**. However, apart from information on the topology of edges, i.e. who is connected to whom, rich time-stamped or sequential data on networks increasingly provides us with higher-order information like, e.g. in which chronological order biological interactions are activated, how users navigate hyperlinks in information networks, or along which paths information propagates in social networks. Previous works in the hosting group have shown that such sequential data contain higher-order, non-Markovian patterns that question edge-centric network analysis - and thus - graph embedding techniques. The lack of graph representation learning techniques that account for higher-order patterns in sequential data on networks has recently been highlighted by researchers in machine learning and network science [5, 6].

Scope of this Project

Addressing the gap summarized above, the goals of this Master thesis are (i) to empirically investigate how higher-order interaction patterns in sequential data on networks influence representation learning techniques, and (ii) to use the higher-order models for paths in networks developed in the hosting group to develop a representation learning algorithm that accounts both for the topology of (dyadic) links in a network as well as patterns in the chronological ordering of links in sequential data. The concrete objectives of the project are as follows:

Objectives

O1 First, the candidate should perform a **literature survey of representation learning in networks**, with a particular focus on vector-space embeddings that use models for paths or walks in networks. The candidate should further **build a collection of implementations of representation learning algorithms** that minimally comprises `node2vec`[3], `Deepwalk`[9] and `HONEM`[11]. For this collection the candidate can use existing and publicly available implementations (e.g. of `node2vec`[a] or `DeepWalk`[b]), while for `HONEM` a custom implementation based on the description of the method in [11] may be neccessary.

O2 Evaluating the methods from **O1**, in a second step the candidate should **develop a probabilistic generative model for paths or walks in a network with a known embedding in a high-dimensional vector space**. The model should generate paths or walks such that the network embedding translates to non-Markovian properties for the generated walks or paths, i.e. for a sequences of nodes $v_0, \ldots, v_l$ traversed by a path of length l the sequence should exhibit "memory" that can be explained based on the known position of traversed nodes/edges in the vector space. This data should then be used to compare the performance of the representation learning techniques identified in **O1**. The expected outcome of this objective is that `HONEM` outperforms methods that do not account for sequential patterns in paths on networks.

O3 The final objective of the thesis is to **apply higher-order models for paths in networks to improve representation learning in sequential network data**. For this, the candidate should investigate how the higher- and multi-order modelling frameworks developed in [2, 7, 10, 12, 13] can be applied to develop an algorithm that generates vector-space embeddings of nodes (or higher-order nodes). The developed algorithm should be evaluated both in the synthetic paths generated in **O2** as well as in real data sets, e.g. performing a cross-validation experiment based on a similarity-based link prediction analogous to [11].

[a] `https://github.com/aditya-grover/node2vec`
[b] `https://github.com/phanein/deepwalk`

The thesis should motivate the problem, describe the current state of research in the field, propose and implement a solution based on prior work on higher-order models, and evaluate it by means of a validation in empirical data. The code of the implementation should be documented in line with typical software engineering standards and this documentation can be included in the appendix of the thesis. The hosting group offers regular meetings in which we will discuss the current progress of the work and clarify potential questions or issues.

Bibliography

[1] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828, 2013.
[2] Christoph Gote, Giona Casiraghi, Frank Schweitzer, and Ingo Scholtes. Mogen: A generative multi-order model to predict variable length paths in networks, 2020.
[3] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864, 2016.
[4] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems*, pages 1024–1034, 2017.
[5] William L Hamilton, Rex Ying, and Jure Leskovec. Representation learning on graphs: Methods and applications. *arXiv preprint arXiv:1709.05584*, 2017.
[6] Renaud Lambiotte, Martin Rosvall, and Ingo Scholtes. From networks to optimal higher-order models of complex systems. *Nature physics*, page 1, 2019.
[7] Timothy LaRock, Vahan Nanumyan, Ingo Scholtes, Giona Casiraghi, Tina Eliassi-Rad, and Frank Schweitzer. Hypa: Efficient detection of path anomalies in time series data on networks, 2019.
[8] Thanh Tam Nguyen and Chi Thang Duong. A comparison of network embedding approaches. Technical report, School of Computer and Communication Sciences, EPFL, Technical Report, 2018.
[9] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: online learning of social representations. In Sofus A. Macskassy, Claudia Perlich, Jure Leskovec, Wei Wang, and Rayid Ghani, editors, *The 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14, New York, NY, USA - August 24 - 27, 2014*, pages 701–710. ACM, 2014.
[10] Martin Rosvall, Alcides V Esquivel, Andrea Lancichinetti, Jevin D West, and Renaud Lambiotte. Memory in network flows and its effects on spreading dynamics and community detection. *Nature communications*, 5:4630, 2014.
[11] Mandana Saebi, Giovanni Luca Ciampaglia, Lance M Kaplan, and Nitesh V Chawla. Honem: Network embedding using higher-order patterns in sequential data. *arXiv preprint arXiv:1908.05387*, 2019.
[12] Ingo Scholtes. When is a network a network?: Multi-order graphical model selection in pathways and temporal networks. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1037–1046. ACM, 2017.
[13] Ingo Scholtes, Nicolas Wider, René Pfitzner, Antonios Garas, Claudio J Tessone, and Frank Schweitzer. Causality-driven slow-down and speed-up of diffusion in non-markovian temporal networks. *Nature communications*, 5:5024, 2014.

Abstract

Representation learning provides crucial input for machine learning algorithms. Learning these representations instead of manually engineering them accelerates the development of ML applications. Various such methods exist for networks, but they mostly rely on first–order Markov chains to generate random walks to explore the network. However, higher–order Markov chains are often better at modeling real–world spreading processes, and this model change may also affect community detection.

We review well–established methods and explore different approaches to upgrade them from first–order to higher–orders. We experiment with multi–class classification and visualization tasks to compare the original and upgraded methods, using an illustrative synthetic grid and real data on social interactions.

The Python source code of the methods and experiments is publicly available[1].

Keywords

node embedding, representation learning, random walk, directed graph, node classification, higher–order network, higher–order Markov model

[1] https://github.com/HONEmbeddings/MScThesis

Acknowledgments

I want to thank my supervisors Prof. Dr. Ingo Scholtes and Vincenzo Perri, for the opportunity to investigate this exciting subject. Moreover, I am very grateful for the excellent preparation, valuable feedback, and pleasant cooperation.

Contents

Chapter 1

Introduction

Networks are essential to model complex systems. However, machine learning algorithms usually cannot use networks directly as input, requiring us to describe the network's nodes through representations (or features) instead. Unless this task is automated, it becomes a bottleneck in an AI application. Hence, network embeddings[1] gained significant interest from researchers and practitioners since 2013 and ongoing.

Most of the embedding methods encode for each node the strengths of the association with the other nodes. Of which LINE [33] is the simplest example because it encodes the edges (i.e., direct neighbors), considering weights if applicable.

Computing an embedding from a network represents a loss of information, which is occasionally desirable if an embedding can generalize (similar to function approximation with neuronal networks). E.g., in social networks, the information about edges is incomplete, and we want to predict unobserved relationships rather than the ones already known. Hence, DeepWalk [23] characterizes a node not only by its direct neighbors but also by its indirect neighbors and therefore uses *random walks* to explore the neighborhood.

Most embeddings assume a classical network, referred to as a first–order network (FON) in this thesis.

However, some networks vary over time. E.g., social interactions (or disease spreads) depend on the chronological sequence of contacts. Such data may be modeled by first extracting paths respecting time constraints and fitting a higher–order Markov model to these paths afterward. We denote networks with random walks generated by a higher–order Markov chain as higher–order networks[2] (HON). According to Rosvall et al. [27], switching to a (more accurate) HON model affects spreading dynamics and community detection, changing the embedding in turn.

To our knowledge, HONEM [28] is the only embedding for higher–order networks. To close this research gap, we analyze the existing embeddings for FON and identify approaches to update them to HON. We compare our HON adaptations of embeddings with their FON counterparts for (multiclass) node

[1] We denote by 'network embeddings' methods that learn vector–space representations for the individual nodes (and possibly tuples of nodes, e.g., edges) of a network.

[2] The term higher-order network (HON) refers exclusively to networks with higher–order Markov chains in this thesis. Other extensions of the FON are mentioned in [17]; see § 2.1.3.

classification and visualization tasks.

1.1 Contributions and Structure

In summary, our contributions are:

- We adapt the FON embeddings DeepWalk [23], Node2vec [10], GraRep [3], and NetMF[25] to HON.
- We compare the performances of these embeddings in node classification tasks using real data.
- We analyze HONEM theoretically and also in the above node classification task.
- We introduce a synthetic network, HON Lattice 2D, and analyze its properties. To distinguish FON and HON embeddings, we changed the HON random walks' dynamic by tweaking the second–order transition probabilities while keeping the FON ones unchanged.

This thesis is organized as follows:

- § 2 explains the relevant background — including HON random walks and the skip-gram model — and introduces the necessary notation. We provide an overview of the existing embeddings and describe some of them. Finally, we present two concepts for later reference: clustering of embedded nodes by parity for lattices and tweaking higher–order probabilities to impact random walks' dynamic.
- § 3 contains our research: identifying different approaches (§ 3.1), our HON variants of embeddings (§ 3.2), and a synthetic network (§ 3.3).
- § 4 evaluates the embeddings in two experiments. § 4.1 applies the embeddings to the synthetic network to investigate theoretic properties and visualize the embeddings. § 4.2 classifies nodes based on social interaction data.
- § 5 summarizes the thesis.
- The appendix contains simulation results, additional experiments, the source code, and development–related communication.
- Finally, a nomenclature lists the definitions used.

Chapter 2

Background

2.1 Networks

A *graph*[1] describes objects (called 'nodes' or 'vertices') and their (*dyadic* i.e. pairwise) relations (called 'edges'). The sets of nodes and edges are denoted by $\mathcal{V}$ and $\mathcal{E}$, respectively. A *network* is a graph that allows for attributes of nodes or edges; examples for attributes are labeled nodes and weighted edges.

Networks are ubiquitous, as they describe social relationships, collaborations, web pages, shopping carts, protein interactions, and transportation of passengers, goods, and energy. To define a particular network, we only need to provide information for each node about the nodes in its neighborhood (i.e., those other nodes connected by an edge with the former node). Examining the network in its entirety, we can then rank nodes by importance, find the shortest paths between nodes, identify communities or bottlenecks, or simulate diseases' transmission. It is incredible how complex systems can be specified by gathering only local information.

We distinguish between *undirected and directed networks*, where the former additionally assumes symmetric relations. In literature, networks are undirected by default. However, I will explicitly mention if a network is assumed to be undirected since this is a restriction compared to the general (i.e., directed) case. Moreover, $(u, v) \in \mathcal{E}$ tests the existence of an edge, regardless of the network's directionality.

2.1.1 Random Walks on Networks

Another crucial concept is a *walk* on a network, essentially a sequence of nodes with the constraint that an edge must exist between two consecutive nodes in the sequence. As this is a fundamental concept for this thesis, let us introduce proper notation. For a tuple of nodes $(s_1, ..., s_L)$ in $\mathcal{V}^L$, use the abbreviation $s_{1..L}$. This tuple is a walk if $(s_\ell, s_{\ell+1}) \in \mathcal{E}$, $\forall 1 \leq \ell < L$, and we use the notation $s_1 \to \ldots \to s_L$ (shortened as $\overrightarrow{s_{1..L}}$).

A *random walk* $S_1 \to \ldots \to S_L$ is a walk with random nodes $(S_1, \ldots, S_L)$, and we denote its probability distribution by $P_{\overrightarrow{S_{1..L}}}(\ldots)$.

[1] As in *graph theory*, see the Seven Bridges of Königsberg problem solved by L. Euler.

The reason for such a rigorous definition is that merely writing $P(A \to B)$ for two nodes A and B conceals the ambiguity between $P_{\overrightarrow{S_{1..2}}}(S_1 = A, S_2 = B)$ and e.g. $P_{\overrightarrow{S_{1..3}}}(S_1 = A, S_2 = B)$. This ambiguity is relevant as the following decomposition is fundamental for random walks:

$$P(A \to B \to C) = P(A) \cdot \underbrace{\frac{P(A \to B)}{P(A)}}_{P(B|A)} \cdot \underbrace{\frac{P(A \to B \to C)}{P(A \to B)}}_{P(C|A \to B)}$$

Hence, we have to assume that $P_{\overrightarrow{S_{1..L}}}$ is **consistent** for different L; this means that $P_{\overrightarrow{S_{1..L}}}$ is the marginal distribution of $P_{\overrightarrow{S_{1..L_{\max}}}}$ for some $L_{\max}$, $\forall L < L_{\max}$. However, in practice, we use the right–hand side of Equation (2.1) to model walk probabilities consistent by construction.

$$P_{\overrightarrow{S_{1..L}}}(S_{1..L} = s_{1..L}) := P_{\overrightarrow{S_1}}(S_1 = s_1) \prod_{\ell=2}^{L} P_{\overrightarrow{S_{1..\ell}}}(S_\ell = s_\ell | S_{1..\ell-1} = s_{1..\ell-1}) \quad (2.1)$$

$$= P_{\overrightarrow{S_{1..L-1}}}(S_{1..L-1} = s_{1..L-1}) \cdot P_{\overrightarrow{S_{1..L}}}(S_L = s_L | S_{1..L-1} = s_{1..L-1}) \quad (2.2)$$

The simulation of random walks is straight forward by using Equation (2.2) iteratively.

2.1.2 First-Order Networks (FON)

An important assumption, known as *first–order Markov property*, states that the edges' distribution depends only on the current node (and not on previously visited ones):

$$P_{\overrightarrow{S_{1..L}}}(S_L = s_L | S_{1..L-1} = s_{1..L-1}) = P_{\overrightarrow{S_{1..L}}}(S_L = s_L | S_{L-1} = s_{L-1})$$

Furthermore, assume the transition probabilities are *time–homogeneous* or *stationary* since otherwise, they would also depend on the current node's index in the sequence:

$$P_{\overrightarrow{S_{1..L}}}(S_L = s_L | S_{L-1} = s_{L-1}) = P_{\overrightarrow{S_{1..2}}}(S_2 = s_L | S_1 = s_{L-1})$$

Define a *first-order network (FON)* as a network with random walks having stationary, first–order Markov transition probabilities, and introduce the notation

$$T_{s \to t} := P_{\overrightarrow{S_{1..2}}}(S_2 = t | S_1 = s), \qquad \forall s, t \in \mathcal{V}$$

which allows simplifying the walk probability of Equation (2.1) to

$$P_{\overrightarrow{S_{1..L}}}(S_{1..L} = s_{1..L}) = P_{\overrightarrow{S_1}}(S_1 = s_1) \prod_{\ell=2}^{L} T_{s_{\ell-1} \to s_\ell}$$

Hence, the random walks in a FON are defined by $P_{\overrightarrow{S_1}}$ and $T_{s \to t}$. An outstanding choice for the former is the *stationary distribution* π, which fulfills $\pi(t) = \sum_{s \in \mathcal{V}} \pi(s) \cdot T_{s \to t}$. While a FON has stationary transition probabilities, we do not require stationarity for the first node of a walk, i.e., $P_{\overrightarrow{S_1}} = \pi$.

Other Definitions I have chosen to introduce FON as a particular case of random walks on a network, but let us also examine the traditional definitions.

Networks allow for edges having attributes, from which we may derive the transition probabilities. By convention, this is the 'weight' attribute, and therefore *weighted networks* are defined as networks, where each edge has a 'weight' attribute. Depending on the context, these edge weights may also store other quantities like strengths of relationship, transport capacities, distances, and time differences. For *unweighted networks*, use an edge weight of one instead.

Accordingly, define the adjacency matrix $\mathbf{A}$ with elements $[\mathbf{A}]_{s,t}$ equal to the weight of the edge $s \to t$ (or zero if this edge does not exist)[2]. The degree matrix $\mathbf{D}$ is diagonal and The transition matrix $\mathbf{T}$ contains the transition probabilities, chosen proportional to the edge weights, $\mathbf{T} = \mathbf{D}^{-1}\mathbf{A}$.

The advantage of using $\mathbf{A}$ over $\mathbf{T}$ for undirected networks is that we can quickly determine the stationary distribution π from $\mathbf{A}$ (using an equilibrium argument) compared to calculating an Eigenvector of $\mathbf{T}$ otherwise.

2.1.3 Higher-Order Networks (HON)

Every model has its limitations. Hence, different approaches for higher–order network models have been considered, which relax the FON's limitations. These involve allowing relations between more than two nodes (simplex, hyperedge), different types of edges, and non–Markovian random walks, see Lambiotte et al. [17].

Instead of abandoning the Markov property altogether, introduce *memory*. More precisely, the K^{th}*–order Markov property* (for some integer $K > 1$) assumes that the transition probabilities $P_{\overrightarrow{S_{1..L}}}(S_L|S_{1..L-1})$ are allowed to depend on the K most recently visited nodes $(s_{L-K}, \ldots, s_{L-1})$ in the walk sequence.

Assuming random walks with a higher–order Markov property is particularly interesting since this may change the random walk's dynamic, see Rosvall et al. [27] and Scholtes et al. [30]. Let us examine the change in dynamic due to adding memory with a transportation network: passengers would have to decide at each station where to go next if no memory (first–order Markov) is assumed; introducing memory allows avoiding returning immediately to the previous station. Hence, first–order random walks are an inefficient way of utilizing some networks.

Stationarity is only required for the (maximal) K^{th}–order transition probabilities, allowing us to rewrite Equation (2.1) for $L > K$:

$$P_{\overrightarrow{S_{1..L}}}(S_{1..L} = s_{1..L}) := \underbrace{P_{\overrightarrow{S_1}}(S_1 = s_1) \prod_{\ell=1}^{K-1} T^{(\ell)}_{s_{1..\ell} \to s_{\ell+1}}}_{P_{\overrightarrow{S_{1..K}}}(S_{1..K} = s_{1..k})} \prod_{\ell=K}^{L-1} T^{(K)}_{s_{\ell+1-K..\ell} \to s_{\ell+1}} \tag{2.3}$$

using the following notation:

$$T^{(k)}_{s_{1..k} \to t} := P_{\overrightarrow{S_{1..k+1}}}(S_{k+1} = t | S_{1..k} = s_{1..k}), \qquad \forall 1 \leq k \leq K$$

Observe that for a HON, the walk's asymptotic dynamics depends entirely on the maximum order transition probabilities ($T^{(K)}_{s_{1..K} \to t}$), while the lower order

[2]Due to the lack of universal convention regarding $[\mathbf{A}]_{s,t}$ referring to the edges $s \to t$ or $t \to s$, I decided for the unambiguous notation $T_{s \to t}$.

ones determine the first K nodes' distribution. Therefore, HON random walks are stationary only for particular choices of $P_{S_1^{\rightarrow}}$ and $T^{(k)}_{s_{1..k}\to t}$ for $k < K$.

Throughout this thesis, a higher–order network model (HON) will denote a network satisfying the higher–order Markov property and stationarity (for the maximum order transition probabilities).

Estimating probabilities Estimating the HON parameters from data is out of scope for this thesis, and we refer to BuildHON+ [38, 39], MON [29], and MoGEN [9] instead; see § 4.2 for an example using MON.

However, unless the random walks are stationary, inconsistencies quickly arise: on the one hand, $T_{s\to t}$ determines the stationary distribution in a FON; on the other hand, $T^{(1)}_{s_1\to t}$ affects only the start of a random walk in a HON. Can we, therefore, convert a HON into a FON by merely dropping the higher–order transition probabilities, or should $T^{(1)}_{s_1\to t}$ differ from $T_{s\to t}$? How about using the first–order stationary distribution π in conjunction with a HON?

Conversely, MoGEN uses a consistent model, including a stop symbol indicating a walk's end. However, this stop represents an absorbing state, leading to a degenerate stationary distribution. (Though, page rank is not affected.)

2.2 Background on Embeddings for FON

Despite the versatility of networks, some tasks require different representations of the information encoded in networks. In particular, networks are not well suited for machine learning tasks, which expect vector–valued data for each node. Hence, extracting suitable features (or representations) is a vital preprocessing step of many machine learning applications. Given this can be a labor–intensive task if done manually, it is no surprise that research about *representation learning* got quite some attention and is indeed an active field of research for a couple of years [2, 13, 40].

When visualizing a network, suitable locations (i.e., coordinates) for the nodes must be chosen. If we want to visualize communities and already have learned a representation suitable for classification, we could apply *t–SNE* to this representation for visualization. If we specify, which nodes should attract or repulse other ones, we can use a force–directed layout. Many popular dimension reduction techniques handle the particular case of pairwise distances (or dissimilarities) between nodes. In particular, PCA [6] and metric MDS [16] aim to find representations in a low–dimensional vector–space while preserving distances between points.

2.2.1 Skip–Gram Model

The *skip–gram model* [20] was introduced for natural language processing (NLP), aiming to predict the neighboring words (denoted as context) for any word. The first step in applying the skip–gram model is to obtain word–context pairs. For example, a sentence (interpreted as a list of words) generates word–context pairs by returning those pairs of words of the sentence, which neither are too far apart nor in the same position.

Let $p(w, c)$ denote the sampling distribution of the word–context pairs. The skip–gram model aims to approximate the conditional distribution $p(c|w)$ of the

context given the word w. However, the $p(c|w)$ values are not stored directly due to the quadratic growth in storage complexity with the number of words and the inability to generalize to unobserved word–context pairs. Instead, embedding vectors $\overrightarrow{w}$ and $\overrightarrow{c}$ for w and c, respectively, are learned such that $p(c|w)$ is approximated by $\hat{p}(c|w)$, which is the softmax (for fixed w and varying c) of the scalar product $\langle \overrightarrow{w}, \overrightarrow{c} \rangle$ between the embeddings of the word w and the context c.

$$\hat{p}(c|w) = \frac{\exp(\langle \overrightarrow{w}, \overrightarrow{c} \rangle)}{\sum_{v \in \mathcal{V}} \exp(\langle \overrightarrow{w}, \overrightarrow{v} \rangle)} \tag{2.4}$$

Note that the embedding usually depends on the role (word or context), so even if w and c are equal, their embeddings $\overrightarrow{w}$ and $\overrightarrow{c}$ might differ.

Hence, two embeddings $\overrightarrow{w}_1$ and $\overrightarrow{w}_2$ for two words w_1 and w_2 have higher cosine similarity, the closer their conditional distributions $p(c|w_1)$ and $p(c|w_2)$ are; see also § 2.3.2.

Training the skip-gram model aims for minimizing the KL-divergence from $\hat{p}(c|w)$ to $p(c|w)$, where the different words use, e.g., $p(w) = \sum_c p(w, c)$ as weights for joint minimization. However, minimizing with stochastic gradient descent (SGD) is not straight forward due to the softmax's denominator (normalization) depending on all context embeddings. Therefore, each gradient descent step must update all of them, which is generally too expensive. There are two approximations of the model provided by Mikolov et al. [21]:

- *Hierarchical softmax* uses a tree structure for the context embeddings, resulting in update costs growing only logarithmically with the number of contexts.
- *Negative sampling*[3] replaces the denominator by a subtrahend consisting of only N samples of contexts with distribution $p_N(c)$. This simplifies updating considerably while still avoiding the unbounded expansion of values, previously achieved by the denominator.

`Word2vec`[4] implements both.

Analytically determining the minimum of the objective of the skip–gram model with negative sampling (SGNS) — see [8, 19] — leads to (see § 3.2.5)

$$\langle \overrightarrow{w}, \overrightarrow{c} \rangle = \log(PMI(w, c)/N) \tag{2.5}$$

where $PMI(w, c) = p(c|w)/p_N(c)$, known as pointwise mutual information (in case p_N were the marginal distribution $\sum_w p(w, c)$). Eq (2.5) is solved by *matrix factorization (MF)* (i.e. singular value decomposition, SVD).

However, matrix factorization can only approximate matrices of rank up to the dimension of the embedding. Hence, both SGNS and matrix factorization define a low–rank approximation of the same matrix, but there is a difference in using weighted or unweighted projections. Another difference arises because matrix factorization substitutes small values of PMI before calculating the logarithm.

In terms of scalability, SGD based methods have an advantage over matrix factorization. Moreover, SGD is easily modified to include penalties or

[3] Negative sampling approximates noise contrastive estimation (NCE), which has a stronger theoretical foundation. Nevertheless, this distinction is not central to this thesis.

[4] `https://code.google.com/archive/p/word2vec/`

constraints. Conversely, matrix factorization is excellent for the theoretical understanding, and there are no parameters (number of iterations and learning rates) affecting convergence.

2.3 Related FON Embeddings

According to the classification of FON embeddings by Hamilton et al. [13], network embeddings correspond to a pair of encoders and decoders. The encoder maps the node of a network to its embedding vector. Training some embedding involves minimizing a loss function, which depends on the decoder. Following this classification, there are three groups of approaches:

- Shallow embeddings
- Autoencoders
- Convolution encoders

Shallow embeddings Shallow embeddings encode nodes by look–up, which requires allocating memory for each embedding vector, but besides this, they are stateless. Decoding generates a scalar value from a pair of embedding vectors.

Many embeddings fall into this category — particularly the ones based on the skip-gram model, which are summarized separately in table 2.1[5].

- Classic embeddings (PCA, MDS, ...)
- Graph Factorization (GF) [1] factorizes (symmetric) adjacency matrices. The distributed calculation allows for massive graphs. However, for embeddings based on adjacency, LINE is preferred.
- GLEE [34] is based on the geometric properties of the Laplacian and scores with its solid theoretical foundation. Nevertheless, requiring an *undirected* network prevents its adaptation to HON.

Autoencoders Autoencoders do not store the individual embedding vectors but calculate them instead. The input is a vector of size $|\mathcal{V}|$ for each node representing the similarities between this node and all network nodes. Function approximation via neuronal networks encodes this vector into a lower dimension before decoding it back to the original size. The state of the autoencoder consists of the parameters of the neuronal network.

This approach comprises two embeddings:

- DNGR [4] approximates the personalized page rank (PPR) similarity, similar to App/Verse.
- SDNE [37] approximates the adjacency vector, similarly to LINE.

[5] Khosla et al. [14] deserve credits for an excellent comparison study, in particular table 2.

Embedding	Word–Context	Train	Symmetric	# Embeddings
DeepWalk [23]	random walk	**HS**	yes	ignore 2nd
LINE–1 [33]	adjacency	NS	yes	one
LINE–2 [33]	adjacency	NS	no	ignore 2nd
PTE [32]	adjacency	NS	no	ignore 2nd
GraRep [3]	multi–step adj.	NCE	no	ignore 2nd
Node2vec [10]	random walk	NS	yes	ignore 2nd
HOPE [24]	similarity	MF	no	two
App [41]	similarity	NS	no	two
Verse [35]	similarity	NCE	no	one
NetMF [25]	random walk	MF	yes / no	ignore 2nd
InfiniteWalk [5]	random walk	MF	yes	ignore 2nd

Table 2.1: The skip–gram model is the basis for many FON embeddings. (LINE–1 is the exception; see § 2.3.2.) The word–context probabilities refer to transitions to adjacent nodes, co–occurrences in random walks, or a similarity measure (e.g., PPR). The embeddings are trained by hierarchical softmax (HS), negative sampling (NS), noise contrastive estimation (NCE), and matrix factorization (MF). DeepWalk, Node2vec, and LINE–1 treat the word–context pairs symmetrically; InfiniteWalk and NetMF (for large window sizes) require undirected networks. Only Verse and LINE–1 use the same embeddings for word and context. The others calculate different embeddings, but only HOPE and App return both. The underlined embeddings will be pursued further.

Convolution encoders Both shallow embeddings and autoencoders are unable to generalize the embedding for new nodes. On the other hand, convolution encoders calculate the embedding while avoiding hardcoding the size of the network $|\mathcal{V}|$, and therefore can embed new nodes because they aggregate features from a neighborhood of each node. Random walks approximate the neighborhood, which might grow huge after just a few steps.

GraphSAGE [12] and various graph convolution networks (GCN) follow this approach.

Considered embeddings After reviewing the different embeddings concerning their potential for generalization from FON to HON (see the approaches in § 3.1), I decided to start with the simple ones. Hence, we will focus on the embedding marked by underlining. Note that GraRep and NetMF cover the second variant of LINE.

Alternatively, GraphSAGE looked promising, and the adaptation of its random walks to HON enjoyable, but its versatility might also require more time to find useful embeddings.

Below, we will explain some FON embeddings in detail.

2.3.1 DeepWalk and Node2vec

DeepWalk [23] was the first embedding for FON to adapt the skip–gram model. The word–context pairs have initially (in NLP) been generated from sentences and are replaced by random walks (of length L) in DeepWalk.

Building on top of Word2vec, DeepWalk generates word–context pairs selecting all pairs of the random walk's nodes within a window of size W (excluding the center of the window). Hence, word and context roles are indistinguishable, which suggests that DeepWalk better suited for *undirected networks*, although there is no hindrance in applying it to *directed networks*. DeepWalk implemented only *unweighted networks* and *hierarchical softmax*. Because DeepWalk adapted to weighted networks and negative sampling corresponds to a special case of Node2vec with parameters $p = 1$ and $q = 1$, I decided to use the adapted variant of DeepWalk instead.

Interestingly, DeepWalk[6] implements a random walk with geometric restart probability (i.e., the personalized page rank, PPR), but this feature was disabled. So, App and Verse would independently suggest embedding the PPR similarity years later.

Node2vec [10] extends DeepWalk and parametrizes the random walk, turning the FON effectively into a 2^{nd}–order network with

$$T^{(2)}_{s_{1..2}\to t} \propto \begin{cases} \frac{1}{p} T_{s_2\to t}, & if\ s_1 = t\ (identical) \\ T_{s_2\to t}, & if\ (s_1, t) \in \mathcal{E}, (adjacent) \\ \frac{1}{q} T_{s_2\to t}, & else \end{cases} \tag{2.6}$$

This adjustment (called bias) provides greater flexibility in adjusting the neighborhood exploration to the task at hand and is motivated by two strategies to traverse trees: breadth–first search (BFS) and depth–first search (DFS).

2.3.2 LINE

Tang et al. [33] did an excellent job explaining the two variants of LINE, and the difference is also crucial for the understanding of the skip–gram model. In particular, we will refer to the clustering of LINE–2 embedded lattices by parity later in this thesis.

For both variants of LINE, the word–context pairs correspond to edges (i.e., adjacent nodes), but they use different approximations to train the embeddings.

The first variant, LINE–1, uses a *different* though related approximation than the skip–gram model. Instead, it embeds the *first–order proximity*, which approximates $p(w, c)$ with (σ is the *sigmoid* function)

$$\hat{p}(w, c) := \sigma(\langle \overrightarrow{w}, \overrightarrow{c} \rangle) = \frac{1}{1 + \exp(-\langle \overrightarrow{w}, \overrightarrow{c} \rangle)}$$

Only undirected networks can be embedded because word and context use the same embeddings. Hence, the approximated probability of an edge between two nodes w and c increases with the scalar product $\langle \overrightarrow{w}, \overrightarrow{c} \rangle$ because σ is strictly monotonically increasing. Therefore, high values of $\langle \overrightarrow{w}, \overrightarrow{c} \rangle$ should relate to adjacent pairs of nodes (w, c).

However, we are more interested in distances than scalar products. If the embedding vectors had equal length, we could conclude that large scalar products correspond to small distances, because $\| \overrightarrow{w} - \overrightarrow{c} \|^2 = \| \overrightarrow{w} \|^2 + \| \overrightarrow{c} \|^2 - 2\langle \overrightarrow{w}, \overrightarrow{c} \rangle$. In figure 2.1, we apply LINE to a network with a two–dimensional lattice structure[7]

[6] https://github.com/phanein/deepwalk

[7] no self–loops; also known as square grid graph

and display distances in the lattice, average lengths, and the angles between pairs of nodes (w_1, w_2). Figure 2.2 displays the corresponding t–SNE visualizations. Admittedly, this required some tuning of t–SNE, but for LINE-1, this worked well.

In contrast, LINE–2 embeds the *second–order proximity*, which approximates $p(c|w)$ with $\hat{p}(c|w) = \exp(\langle \overrightarrow{w}, \overrightarrow{c} \rangle)/const(w)$; see § 2.2.1. So does this preserve the lattice structure too? No, according to figure 2.2, the embedded lattice structure consists of two separate clusters, and according to figure 2.1, that the scalar product $\langle \overrightarrow{w_1}, \overrightarrow{w_2} \rangle$ of a pair of nodes (w_1, w_2) is generally bigger if the distance between w_1 and w_2 is two (compared to one or three). Pairs of nodes with (shortest path) distance two are also the only ones with overlapping neighborhoods.

Define a node's **parity** (for a lattice) as the parity bit of the sum of its coordinates. Hence, pairs of nodes with overlapping neighborhoods must have the same parity. Intuitively, each embedded node $\overrightarrow{w}$ is pulled towards all of its neighboring contexts' embeddings $\overrightarrow{c}$, while the negative samples act as a counterforce. It appears that the nodes with the same parity are (indirectly) linked together, while there are no such connections otherwise.

Embeddings of lattices are the primary model for investigating the theoretical properties of embeddings throughout this thesis. Moreover, explaining the clustering by parity was the primary rationale for explaining LINE in detail.

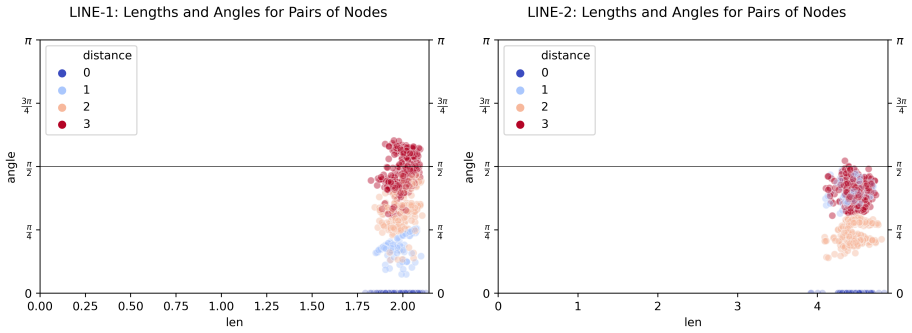


Figure 2.1: Examine the scalar product $\langle \overrightarrow{w_1}, \overrightarrow{w_2} \rangle = len^2 \cos(angle)$ for a pair of nodes (w_1, w_2) for LINE–1 (left) and LINE–2 (right). The 'distance' refers to the shortest path distance in the lattice. For distance zero, observe the concentration of the lengths $\|\overrightarrow{w}\|$, which confirms the analogy between scalar product $\langle \overrightarrow{w_1}, \overrightarrow{w_2} \rangle$ and distance $\|\overrightarrow{w_1} - \overrightarrow{w_2}\|$. For LINE–1, the angle between pairs of nodes grows with their distance, indicating that the embedding might reflect the lattice structure. For LINE–2, the angles are considerably smaller for distance two (than one or three), indicating that these are the closest neighbors in the embedding space.

2.3.3 GraRep

GraRep [3] takes a middle ground between embedding the adjacency matrix (LINE–2) and a random walk (DeepWalk) by embedding multi–step transition probabilities separately for each step–size. However, it differs from the others by training the embedding with matrix factorization instead of SGD.

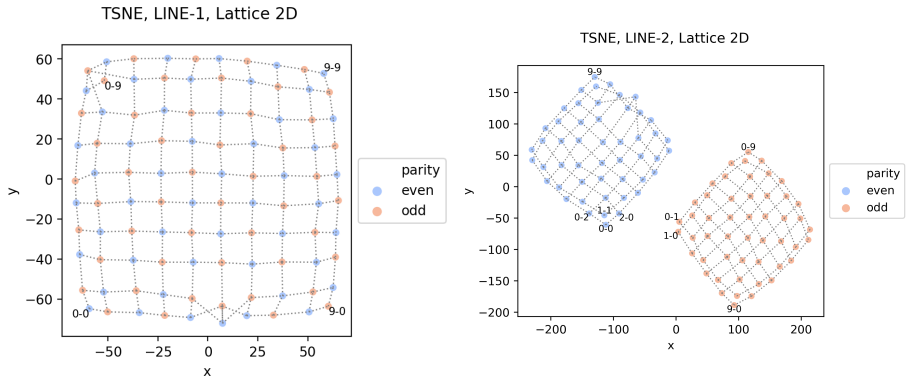


Figure 2.2: T–SNE visualizations of embeddings of a two-dimensional lattice with LINE–1 (left) and LINE–2 (right). The colors signify the parity bit of the sum of the coordinates of each node. LINE–1 mostly preserves the lattice structure, which is further highlighted by connecting adjacent nodes. The LINE–2 embeddings fall into two groups, the parity, and connecting adjacent edges would not be helpful. Instead, horizontal and vertical two-step neighbors are connected.

Hence, for each step–size $k \in \{1, \ldots, S\}$ define

$$PMI^{(k)}(w, c) = \frac{[\mathbf{T}^k]_{w,c}}{\frac{1}{|\mathcal{V}|}\sum_{u \in \mathcal{V}}[\mathbf{T}^k]_{u,c}}$$

and factor $\langle \overrightarrow{w}, \overrightarrow{c} \rangle = \max(0, \log(PMI^{(k)}(w, c)/N))$ with truncated SVD to obtain $\mathbf{U}_R^{(k)} \cdot \mathbf{\Lambda}_R^{(k)} \cdot \mathbf{V}_R^{(k)}$, where R is the embedding dimension, and N is the number of negative samples[8]. Finally, concatenate the individual word embeddings $\mathbf{U}_R^{(k)} \cdot \sqrt{\mathbf{\Lambda}_R^{(k)}}$ after *scaling*[9] them.

Note that when applying GraRep to a lattice, we observe the same clustering by parity issue as LINE–2. This happens due to not aggregating different step–sizes.

The authors of GraRep argue that aggregating over different step-sizes mixes up distinct features. While this argument initially sounds convincing, we should nevertheless remember that flexibility is a double–edged sword. Maybe a parameterized aggregation of different step–sizes before the matrix factorization would be useful — in the same way as Node2vec extends DeepWalk.

2.3.4 NetMF and InfiniteWalk

NetMF [25] adapts DeepWalk by using the connection between SGNS and matrix factorization discovered by Levy and Goldberg [19]. It assumes an *undirected, weighted network*, enabling the use of Eigendecomposition and a closed–form expression for the stationary distribution π proportional to the degree.

The distribution of word–context pairs corresponds to *co–occurrence probabilities of random walks* starting at the word–node and visiting W context–

[8] https://github.com/ShelsonCao/GraRep has $N = 1$ hardcoded.
[9] According to the source code. The paper does not mention this.

nodes. The length of the random walk W corresponds to the *window size* in DeepWalk and is therefore named the same.

$$p(c|w) = \frac{1}{W}\sum_{k=1}^{W}[\mathbf{T}^k]_{w,c} \tag{2.7}$$

NetMF implicitly assumes that the marginal distribution $p(w)$ equals the stationary distribution π, justifying the distribution of negative samples $p_N = \pi$ because the other marginal distribution $p(c)$ is the same. Conversely, DeepWalk weights the start of random walks uniformly, and $p(w)$ is somewhere 'in between' the uniform (for short L) and the stationary (for large L) distribution. Also, GraRep assumes a uniform distribution for $p(w)$ because otherwise, $PMI^{(k)}$ denominator would need to be a weighted average. Before NetMF, the authors of GraRep also introduced a similar method, E–SGNS, which differs from NetMF by the chosen p_N.

For *small window sizes* W, we directly calculate $PMI(w,c) = p(c|w)/p_N(c)$ using Equation (2.7) and factor $\langle\overrightarrow{w}, \overrightarrow{c}\rangle = \max(0, \log(PMI(w,c)/N))$ with truncated SVD of dimension R to obtain $\mathbf{U}_R \cdot \mathbf{\Lambda}_R \cdot \mathbf{V}_R$, where N is the number of negative samples. Finally, the resulting word embeddings are $\mathbf{U}_R \cdot \sqrt{\mathbf{\Lambda}_R}$.

Approximation For *large window sizes* W, Equation (2.7) is computationally challenging; hence we utilize the symmetry of the adjacency matrix $\mathbf{A}$ of an *undirected* network. Define the *symmetrized transposition matrix* $\widetilde{\mathbf{T}} = \mathbf{D}^{-0.5}\mathbf{A}\mathbf{D}^{-0.5}$ and use the Eigendecomposition $\widetilde{\mathbf{T}} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^{-1}$. This simplifies the calculation of powers of $\mathbf{T} = \mathbf{D}^{-0.5}\mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^{-1}\mathbf{D}^{+0.5}$, and hence

$$p(c|w) = \mathbf{D}^{-0.5}\mathbf{Q}\Big(\frac{1}{W}\sum_{k=1}^{W}\mathbf{\Lambda}^k\Big)\mathbf{Q}^{-1}\mathbf{D}^{+0.5} \tag{2.8}$$

Truncating the Eigendecomposition (as in 'truncated SVD') further simplifies the calculation, and afterward proceed as above.

InfiniteWalk The limit of NetMF for $W \to \infty$ is investigated by InfiniteWalk [5], which also assumes an undirected network to utilize Equation (2.8).

For this thesis, we have to assume directed networks, and NetMF with small window sizes is easily adapted and does not enforce symmetry (unlike DeepWalk). However, for NetMF with large window sizes and InfiniteWalk, this cannot be fixed. Therefore, we do not pursue InfiniteWalk further.

2.4 Related HON Embeddings

To our knowledge, HONEM [28] is the only published embedding for HON. Building on top of BuildHON+ [38, 39], it aggregates higher–order transition probabilities into a neighborhood matrix $\mathbf{D}_{\text{neighbor}}$, which is factored by SVD to obtain the embedding. However, given the vast number of FON embeddings, there is certainly still room for more competition.

2.4.1 Definition of HONEM

After fitting a K^{th}–order Markov model (including estimation of K) with BuildHON+, we obtain transition probabilities ('rules') $T^{(k)}_{s_{1..k}\to t}$ for any tuple $s_{1..k}$ corresponding to a walk $\overrightarrow{s_{1..k}}$ ($k <= K$). However, some tuples are unavailable due to the lack of data (i.e., walk $\overrightarrow{s_{1..k}}$ has probability zero) or the removal of unnecessary transition probabilities ('rule pruning'). Hence, $T^{(k)}_{s_{1..k}\to t}$ is only available for $s \in rule\_keys$, with

$$rule\_keys := \bigcup_{k=1}^{K} \{s_{1..k} | \exists\, T^{(k)}_{s_{1..k}\to t}\} \subseteq \bigcup_{k=1}^{K} \{s_{1..k} \in \mathcal{V}^k \mid \overrightarrow{s_{1..k}} \text{ is a walk}\}$$

The neighborhood matrix $\mathbf{D}_{\text{neighbor}}$ is defined as

$$\mathbf{D}_{\text{neighbor}} := \sum_{k=1}^{K} \exp(1-k) \cdot \mathbf{D}^{(k)}_{\text{neighbor}}$$

based on the *k^{th}–order neighborhood matrix* $\mathbf{D}^{(k)}_{\text{neighbor}}$ — with $average(\emptyset) = 0$

$$[\mathbf{D}^{(k)}_{\text{neighbor}}]_{i,j} := average\big(\{T^{(k)}_{s_{1..k}\to t} \mid s \in rule\_keys, s_1 = i, t = j\}\big)$$

Note that $\mathbf{D}^{(1)}_{\text{neighbor}}$ corresponds to the first–order transition matrix $\mathbf{T}$.

Applying truncated SVD with dimension R to $\mathbf{D}_{\text{neighbor}}$ provides the factorization $\mathbf{U}_R \cdot \mathbf{\Lambda}_R \cdot \mathbf{V}_R$, and the embedding is defined as $\mathbf{U}_R \cdot \sqrt{\mathbf{\Lambda}_R}$.

2.4.2 Theoretical Analysis of HONEM

Before addressing the weaknesses of HONEM, let us keep in mind that BuildHON+ decided to prune the rules to reduce memory requirements. Similarly, the HONEM neighborhood matrix's construction is computationally cheap, making HONEM suitable for large graphs. (As long as truncated SVD is feasible.)

First, observe that HONEM does not introduce any non–linearity (i.e., log) before factorizing the neighborhood matrix. However, InfiniteWalk [5] claims that embeddings using a low–dimensional matrix factorization without non–linearity perform poorly compared to skip–gram methods, which factorize the logarithm of (positive, shifted) PMI.

From a statistical modeling point of view, however, the construction of the neighborhood matrix — mainly the higher–order terms — is unsatisfactory, as it **averages** the transition probabilities of the **last**–step over walks of a given length. (Moreover, are walks with $T^{(k)}_{s_{1..k}\to t} = 0$ or ≈ 0 included in the average?) The **sum** of the conditional (on the first node only!) probabilities of any walk ending at a given point would be better interpretable. However, if we also follow the advice of GraRep [3] to avoid aggregating multi–step transition probabilities of different orders, we would end up with HON GraRep; see § 3.2.4.

The rule pruning of BuildHON+ particularly has a detrimental effect on the HONEM neighborhood matrix: While the neighborhood matrix depends continuously on the output of BuildHON+, the pruning mechanism introduces a discontinuity, allowing the design of adversary examples — which is fun and informative. Inspired by [30], we construct a HON where the higher–order

probabilities are adjustable without affecting the first–order terms. Moreover, this is an excellent opportunity to introduce an essential technique for this thesis.

Figure 2.3 illustrates the effect of pruning on a simple network consisting of three nodes A, B, and C. The first–order transition probabilities are $T_{A\to B} = 1$, $T_{B\to A} = 0.5 = T_{B\to C}$, and $T_{C\to B} = 1$. For $\omega \in (-0.5, 0.5)$, define $T^{(2)}_{AB\to C} = 1 + \omega = T^{(2)}_{CB\to A}$ and $T^{(2)}_{AB\to A} = 1 - \omega = T^{(2)}_{CB\to C}$. For $\omega > 0$, the walks $A \to B \to C$ and $C \to B \to A$ become more likely, which we denote as *speed–up*. Conversely, $\omega < 0$ results in a *slow–down*. Note that the first–order stationary distribution π is not affected by this change. ($\pi(B) = 0.5$ remains unchanged because a walk hast to visit node B every second step, and $\pi(A) = \pi(C)$ follows from symmetry.)

The (A, C)–entry of the second–order neighborhood matrix, $[\mathbf{D}^{(2)}_{\text{neighbor}}]_{A,C}$ equals to $T^{(2)}_{AB\to C}$, unless this rule is pruned by BuildHON+ (i.e., $\omega \approx 0$ or equivalently $T^{(2)}_{AB\to C} \approx T_{B\to C}$) resulting in $[\mathbf{D}^{(2)}_{\text{neighbor}}]_{A,C} = 0$. Note, that both *slow–down* ($\omega < 0$) and *speed–up* ($\omega > 0$) result in $[\mathbf{D}^{(2)}_{\text{neighbor}}]_{A,C} > 0$. This is unfortunate because tricks to save memory should have (almost) no impact on the embedding. Consequently, throughout this thesis, **HONEM is calculated without rule pruning**.

However, accomodating for the rule pruning would make HONEM's computation more expensive and give up its niche.

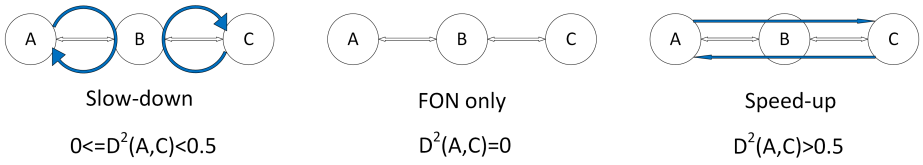


Figure 2.3: Effect of rule pruning on the second–order neighborhood matrix $\mathbf{D}^{(2)}_{\text{neighbor}}$. The blue arrows indicate second–order transition probabilities exceeding corresponding first–order ones. The only walk from A to C of length 2 is $A \to B \to C$, which has probability $\pi(A) \cdot T_{A\to B} \cdot T^{(2)}_{AB\to C}$ and therefore is *slowed–down* or *speeded–up* if $T^{(2)}_{AB\to C} < 0.5$ or > 0.5, respectively. By definition, $[\mathbf{D}^{(2)}_{\text{neighbor}}]_{A,C} = T^{(2)}_{AB\to C}$, but if $T^{(2)}_{AB\to C} \approx T_{B\to C}$, BuildHON+ might prune this rule resulting in $[\mathbf{D}^{(2)}_{\text{neighbor}}]_{A,C} = 0$. This *discontinuity* is undesirable.

Chapter 3

Methodology

3.1 Approaches for HON embeddings

After gaining an overview of existing embeddings for first–order networks, let us see how to adapt these to higher-order networks. Here, we lay out the ideas, while § 3.2 describes the embeddings considered.

3.1.1 Higher-order random walks

Given that random walks make the difference between first– and higher–order networks, adapting the random walks is the most natural approach to generalizing FON embeddings to HON. Conveniently, no changes to the down-stream task are necessary.

Hence, let us identify the FON embeddings of § 2.3 depending on random walks, starting with embeddings that *sample* random walks:

- DeepWalk generates word-context pairs via random walks.
- Node2vec generates word-context pairs via biased random walks.
- App and Verse use the personalized page rank (PPR) similarity, defined as a random walk with geometric reset probability.
- GraphSAGE uses random walks to sample the neighborhood of each node.

Other embeddings calculate the *distribution of nodes visited* by random walks:

- GraRep embeds k–step transition probabilities, which are the distribution of the endpoint of random walks truncated to length k.
- NetMF is the matrix factorization variant of DeepWalk.
- DNGR encodes the personalized page rank (PPR) similarity.

These embeddings all lead to viable candidates for HON embeddings, and four will be pursued further in § 3.2.

However, while the adaptation of random walks from FON to HON is straightforward mathematically, it nevertheless poses computational challenges. A symmetric FON adjacency matrix $\mathbf{A}$ enables calculating the distribution of nodes quickly using approximation (2.8). However, this trick does not work for directed and higher-order networks; see algorithm 3.1 instead.

3.1.2 Beyond pairwise interactions

All shallow embeddings in § 2.3 do model only *pairwise interactions*. We are concerned that this might be *too restrictive* because we cannot recover a multivariate distribution (e.g., the stationary distribution of the K^{th}–order Markov model) from bivariate marginal distributions. However, the multivariate distribution also has higher complexity than its marginals.

Moreover, the down–stream task may prevent embedding higher interactions, as it happens for this thesis.

Embedding higher–order interactions is straight forward: Each K^{th}–order Markov chain on a network corresponds to a first–order Markov chain with K–tuples of nodes[1] as state–space. Hence, we may use tuples of nodes (or walks) as words and contexts (instead of single nodes). To avoid redundancy, we could also assign walks as word and nodes as context.

This leaves us with three options to model interactions:

Pairs of Nodes Both words and contexts are single nodes. To utilize the HON, we combine this with HON random walks; see § 3.1.1. Inevitably, this cannot capture the full complexity of a HON model.

Mixed The word is a walk, and the context is a single node. Unlike the previous case, we can store the entire HON transition probabilities $T^{(k)}_{w_{1..k}\to c}$ for $k \leq K$ in the conditional word–context distribution $p(c|w)$. Similarly, we can distinguish different starts (e.g., $(s_1), (s_1, s_2)$) of random walks. This flexibility also comes at a (storage) cost.

Pairs of Walks Both words and contexts are walks. This is even more flexible than the 'mixed' case but also includes redundant information. (Not investigated further.)

To decrease the cost of embedding walks in the 'mixed' case, I considered two approaches:

- Exploiting the start–tuples' hierarchical structure $(s_K) \subset s_{K-1..K} \subset \ldots \subset s_{1..K}$ and introducing a penalty on $\sum_{\ell=1}^{K-1} \|\overrightarrow{s_{\ell+1..K}} - \overrightarrow{s_{\ell..K}}\|^2$ allows reducing the number of samples for training; see § 3.2.5.

- Instead of embedding each walk independently, combine the walk's individual nodes' embeddings with a Hadamard product. This reduces the embeddings' storage, and leads to *Tensor decomposition* [15], brought to my attention by HOSGNS [24]. (Not pursued further.)

The HON embeddings in § 3.2 implement both 'pairs of nodes' (four times) and 'mixed' (three times) interactions.

However, the down–stream task ultimately decides whether higher–order interactions are beneficial. For the classification task in § 4.2, we learn labels for nodes, therefore learning embeddings for tuples of nodes is pointless, and we use only 'pairs of nodes' interactions.

[1] De Bruijn graph

3.2 Higher–order network embeddings examined

This section describes our HON embeddings, which all base on the skip–gram model (unlike HONEM). We simulate HON random walks iteratively using Equation (2.3) or equivalently:

$$P_{\overrightarrow{S_{1..L+1}}}(S_{L+1} = s_{L+1} | S_{1..L} = s_{1..L}) = \begin{cases} P_{\overrightarrow{S_1}}(S_1 = s1) & if\ L = 0 \\ T^{(L)}_{s_{1..L} \to s_{L+1}} & if\ 1 \leq L \leq K \\ T^{(K)}_{s_{L+1-K..L} \to s_{L+1}} & else \end{cases}$$

Note that by tweaking the network, we can add functionality, such as FON variants of embeddings, cross–validation, or the biased random walk of Node2vec.

All embeddings implement a `decode`–method for calculating $\hat{p}(c|w)$ from the embedding. We may improve this approximation by consulting the network topology (`use_neighborhood`), excluding self-loops (`no_self_loops`), and normalizing to one (`normalize`).

3.2.1 HON DeepWalk

HON DeepWalk combines HON random walks with Word2vec and therefore inherits the restriction that the word–context pairs are treated symmetrically. Consequently, it only supports 'pair of nodes' interactions.

The parameters for generating the HON random walks are `num_walks` (number of random walks per node) and `walk_length` L (length/number of steps of each random walk). The parameters for Word2vec are `dimension` R (dimension or size of the embedding), `window_size` W, `hs` (use hierarchical softmax), `negative` N (number of negative samples), `num_iter` (number of iterations), plus others. The `random_seed` ensures that random walks and embedding are reproducible.

As mentioned in § 2.3.1, I prefer negative sampling, even though FON DeepWalk uses hierarchical softmax.

3.2.2 HON Node2vec

Node2vec introduced a bias (with parameters p and q) affecting all random steps except the first one. This bias perfectly fits into the HON framework and is implemented by tweaking the HON transition probabilities.

For HON Node2vec, first, ensure that the network contains at least the second–order probabilities $T^{(2)}_{s_{1..2} \to t}$, $\forall (s_1, s_2) \in \mathcal{E}$, and define $T^{(2)}_{s_{1..2} \to t} := T_{s_2 \to t}$ otherwise. Second, replace $T^{(k)}_{s_{1..k} \to t}$ with $T^{(k),bias}_{s_{1..k} \to t}$ for $k > 1$ with

$$T^{(k),bias}_{s_{1..k} \to t} \propto \begin{cases} \frac{1}{p} T^{(k)}_{s_{1..k} \to t}, & if\ s_{k-1} = t\ (identical) \\ T^{(k)}_{s_{1..k} \to t}, & if\ (s_{k-1}, t) \in \mathcal{E}, (adjacent) \\ \frac{1}{q} T^{(k)}_{s_{1..k} \to t}, & else \end{cases}$$

Finally, embed the modified network with HON DeepWalk.

3.2.3 HON NetMF

HON NetMF generates word–context pairs corresponding to HON random walks and embeds them with matrix factorization. It implements both 'pairs of nodes' and 'mixed' interactions.

For the distribution of the negative samples p_N, we use the stationary distribution π. For FON, this is natural because it follows from assuming $p(w) = \pi(w)$. For HON, however, this would require stronger assumptions than just stationarity of the maximal order $T^{(K)}_{s_{1..K} \to t}$; see the remarks in § 2.1.3 on estimating probabilities.

The parameters are `dimension` R (dimension of the embedding), `negative` N (number of negative samples), `pairwise` (use 'pairs of nodes' instead of 'mixed' interactions, default=`False`), and `window_size` W (number of steps of random walk).

Optimization While $p(c|w)$ is the distribution of nodes appearing in a random walk starting with w, we should not enumerate all these walks to calculate the probabilities. Because under the weak assumptions that each node has an out–degree of at least two, the number of walks *grows exponentially* with the number of steps W.

Note that FON NetMF (for small window sizes) calculates $\sum_{k=1}^{W} \mathbf{T}^k$ sequentially, using $\mathbf{T}^k = \mathbf{T}^{k-1} \cdot \mathbf{T}$, and therefore in linear time. How about HON? For a HON of order K, simulating another step for a random walk requires only knowledge of the K most recently visited nodes. To calculate $p(c|w)$ for a HON in *linear time*, $O(W)$, and *constant space*, we calculate the k–step distributions sequentially and still enumerate walks but have to *truncate* them after each step. The finite number of states of the HON bounds the number of different walks required to calculate the next step; see algorithm 3.1.

The algorithm also works for variable–order Markov models (e.g., BuildHON+), provided the keys $s_{1..k}$ fulfill:

$$s_{1..k} \in \texttt{rules} \implies s_{2..k} \in \texttt{rules}$$

Credits go to the minibatch algorithm of GraphSAGE [12, § A], which inspired algorithm 3.1.

3.2.4 HON GraRep

The HON GraRep embedding calculates the multi-step transition probabilities congruent with HON random walks, embeds them with matrix factorization, and utilizes a similar optimization as HON NetMF.

The parameters are `num_steps` S (number of steps), `dimension` R (dimension of the embedding of a single step; the total dimension is $R \cdot S$), `negative` N (number of negative samples), `pairwise` (use 'pair of nodes' instead of 'mixed' interactions, default `True`), `neg_stationary` (use stationary distribution instead of uniform for p_N, default `False`), `normalize` (scale the individual embeddings before concatenating them, default `False`).

I introduced the parameters `neg_stationary` and `normalize` for comparison with HON NetMF and had to turn the latter off due to poor performance together with LogisticRegression in § 4.2. Whether FON GraRep indeed uses `normalize=True` is dubious; see § 2.3.3.

```
# transition probabilities are stored in a dictionary
rules[s_{1..k}] = { t : T^(k)_{s_{1..k}→t} }

def find_rule_key(start): # start is tuple of nodes
    # Find longest key such that start ends with key.
    # Length of this key is at most the max. order K.
    return sorted([ key for key in rules.keys()
                    if key==start[-len(key):] ]
                  ,key=len)[-1]

def probability_of_context_given_word(start, num_steps):
    pc_w = defaultdict(float)
    walk_probs = { start : 1 }
    for step in range(num_steps):
        walk_probs_new = defaultdict(float)
        for w,pw in walk_probs.items():
            for t,pt in rules[find_rule_key(w)].items():
                p = pw * pt
                pc_w[t] += p / num_steps
                w_new = (*w, t) # append t to w
                w_new = find_rule_key(w_new) # truncate
                walk_probs_new[w_new] += p
        walk_probs = walk_probs_new
    return pc_w
```

Algorithm 3.1: The optimized calculation for HON NetMF determines the frequency $p(c|w)$ of nodes c visited in a random walk when starting with a provided tuple of nodes w. The complexity of enumerating all walks generally grows *exponentially* with the number of steps. Utilizing the maximal order K of the Markov model allows truncating the walks, resulting in *linear* complexity.

3.2.5 HON Experimental Embedding

Finally, this experimental embedding implements a HON LINE-2 with 'mixed' interactions and a penalty on the walk hierarchy. Its training uses SGD and negative sampling without relying on Word2vec (due to its limitations). The 'mixed' interactions are required because a HON LINE–2 with 'pairs of nodes' interactions would only capture the first–order probabilities. The start nodes are shuffled, and the negative samples' distribution p_N is either uniform or the stationary distribution π. The embeddings of word and context are separate. **This embedding primarily serves to test ideas**, such as the penalty; see below. Without the penalty, it allows comparing SGD against NetMF's matrix factorization (for $W = 1$). It is principally a starting point to implement HON App/Verse, but its performance limits its use to a proof–of–concept. (The performance tuning of gensim's Word2vec implementation is explained in a blog, see [26].)

The parameters are `dimension` R (dimension of the embedding), `steps` (number of iterations per start path), `negative` N (number of negative sam-

ples), `penalty` τ (penalty parameter), `neg_stationary` (use stationary distribution instead of uniform for pN, default `True`), `learning_rate_start` and `learning_rate_end` (learning rates are linear interpolated between start and end), `seed` (random seed), `max_start_len` (train only short start tuples).

Hierarchical structure of walks Observe that the start–tuples of a walk follow a hierarchical structure $(s_K) \subset s_{K-1..K} \subset \ldots \subset s_{1..K}$, and a larger tuple indicates more information about the next node. If the transition probabilities were *stationary*, $T^{(k)}_{s_{1..k}\to t} = P_{\overrightarrow{S_{\ell+1..\ell+k+1}}}(S_{\ell+k+1} = t | S_{\ell+1..\ell+k} = s_{1..k}), \forall \ell \geq 0$, for *all orders* $k \leq K$, $T^{(k-1)}_{s_{2..k}\to t}$ would be a weighted average of $T^{(k)}_{s_{1..k}\to t}$, because

$$\underbrace{P_{\overrightarrow{S_{2..k+1}}}(S_k + 1 = t | S_{2..k} = s_{2..k})}_{T^{(k-1)}_{s_{2..k}\to t}, \text{ if stationary}} = \sum_{s_1} T^{(k)}_{s_{1..k}\to t} \cdot P_{\overrightarrow{S_{1..k+1}}}(S_1 = s_1 | S_{2..k} = s_{2..k})$$

Which is also confirmed by figure 4.5.

Since 'mixed' interactions require training embeddings for more words, this also needs more word–context pairs. As a countermeasure, we add a penalty on $\overrightarrow{s_{k-1..K}} - \overrightarrow{s_{k..K}}$ for $k \in \{2, \ldots, K\}$ to SGNS's loss function to utilize the dependency between $T^{(k-1)}_{s_{2..k}\to t}$ and $T^{(k)}_{s_{1..k}\to t}$.

In § 4.2.5, paragraph 'entropy of transition probabilities', we expand on the ideas presented here; see also figure 4.12.

Gradient of Loss To calculate the gradients for minimization by SGD, we follow the derivation of Levy and Goldberg [19]. They define the loss of a single observed word–context pair (w, c) as (where σ is the *sigmoid function*)

$$-\Big(\log \sigma(\langle \overrightarrow{w}, \overrightarrow{c} \rangle) + N \sum_{c_N} p_N(c_N) \log \sigma(-\langle \overrightarrow{w}, \overrightarrow{c_N} \rangle) \Big)$$

Since we want to add a penalty term to the loss function, we average rather than sum these individual losses to make the loss $\mathcal{L}$ independent from the total number of pairs. Hence, we replace their number of word–context pairs, $\#(w, c)$, with probabilities, $p(w, c)$.

$$\begin{aligned}
\mathcal{L} &:= \mathcal{L}_{penalty} + \sum_w \sum_c \mathcal{L}(w, c) \\
\mathcal{L}(w, c) &:= -\big(p(w, c) \log \sigma(\langle \overrightarrow{w}, \overrightarrow{c} \rangle) + N \cdot p(w) \cdot p_N(c) \log \sigma(-\langle \overrightarrow{w}, \overrightarrow{c} \rangle) \big) \\
\mathcal{L}_{penalty} &:= \tau \sum_{(w_1, w_2) \in \mathcal{H}} \left\| \overrightarrow{w_1} - \overrightarrow{w_2} \right\|^2
\end{aligned}$$

For the penalty term $\mathcal{L}_{penalty}$, we introduced a penalty parameter $\tau \geq 0$ and defined $\mathcal{H} = \bigcup_{k=2}^{K} \{(s_{1..k}, s_{2..k}) | s_1 \to \ldots \to s_k \text{ is a walk}\}$, the set of pairs of walks with a child–parent relation according to the walk hierarchy.

To calculate the gradients, set $x = \langle \overrightarrow{w}, \overrightarrow{c} \rangle$ and use $\partial \log \sigma(x)/\partial x = \sigma(-x)$

$$\nabla_{\overrightarrow{w}} \mathcal{L} = \nabla_{\overrightarrow{w}} \mathcal{L}_{penalty} + \sum_{c} \nabla_{\overrightarrow{w}} \mathcal{L}(w, c)$$

$$\nabla_{\langle \overrightarrow{w}, \overrightarrow{c} \rangle} \mathcal{L}(w, c) = -\big(p(w, c) \cdot \sigma(-\langle \overrightarrow{w}, \overrightarrow{c} \rangle) - N \cdot p(w) \cdot p_N(c) \cdot \sigma(+\langle \overrightarrow{w}, \overrightarrow{c} \rangle)\big)$$

$$\nabla_{\overrightarrow{w}} \mathcal{L}(w, c) = \nabla_{\langle \overrightarrow{w}, \overrightarrow{c} \rangle} \mathcal{L}(w, c) \cdot \overrightarrow{c}$$

$$\nabla_{\overrightarrow{w}} \mathcal{L}_{penalty} = \tau \sum_{(w_1, w_2) \in \mathcal{H}} \big(\overrightarrow{w_1} - \overrightarrow{w_2}\big) \cdot \begin{cases} 2 & \text{if } w = w_1 \\ -2 & \text{if } w = w_2 \\ 0 & \text{else} \end{cases}$$

$$\nabla_{\overrightarrow{c}} \mathcal{L} = \sum_{w} \nabla_{\overrightarrow{v}} \mathcal{L}(w, c)$$

$$\nabla_{\overrightarrow{c}} \mathcal{L}(w, c) = \nabla_{\langle \overrightarrow{w}, \overrightarrow{c} \rangle} \mathcal{L}(w, c) \cdot \overrightarrow{w}$$

While we are at it, let us quickly verify Equation (2.5) for $\tau = 0$ (as promised in § 2.2.1). The minimum condition is (using $\sigma(x) = \exp(x)\sigma(-x)$)

$$0 \stackrel{!}{=} \frac{\nabla_{\langle \overrightarrow{w}, \overrightarrow{c} \rangle} \mathcal{L}(w, c)}{\sigma(-\langle \overrightarrow{w}, \overrightarrow{c} \rangle)} = -\big(p(w, c) - N \cdot p(w) \cdot p_N(c) \cdot \exp(\langle \overrightarrow{w}, \overrightarrow{c} \rangle)\big)$$

which leads to the known result

$$\exp(\langle \overrightarrow{w}, \overrightarrow{c} \rangle) = \frac{p(w, c)}{N \cdot p(w) \cdot p_N(c)} = \frac{PMI(w, c)}{N}$$

3.2.6 HONEM

Being the only HON embedding published, this is the benchmark to compare other embeddings with. This implementation calculates the neighborhood matrix $\mathbf{D}_{\text{neighbor}}$ from HONEM and generates an embedding using matrix factorization.

While the original HONEM [28] used BuildHON+ (a variable order model) to estimate the transition probabilities, we favored using the same probabilities for all embeddings to facilitate comparison. So our implementation of HONEM is a fixed order model then, though technically it supports both fixed and variable order transition probabilities. However, this is no disadvantage as the pruning of rules in BuildHON+ causes inconsistencies in HONEM; see § 2.4.2.

3.3 Synthetic network (HON Lattice 2D)

Let us introduce the HON Lattice 2D, a synthetic network for the comparison of embeddings.

In § 2.3.2, we already embedded a two-dimensional lattice with both LINE variants to understand the embeddings' structure. We will see in § 4.1 that other embeddings also preserve the lattice structure. Combine this with the tweaking of higher–order terms to slow–down or speed–up walks from § 2.4.2. By construction, HON aware embeddings are affected by this kind of dynamic change, while the FON embeddings (see § 2.3) are not.

The **HON Lattice 2D** is a higher–order network with a parameter $\omega \in (0, 1)$ controlling the higher–order dynamic, which speeds up horizontal movements and slows down vertical movements, as illustrated in figure 3.1. (The first–order lattice is equivalent to $\omega = 0$.) Speed–up means moving twice in the same direction has increased probability at the expense of moving in the opposite direction, while other movements are unchanged. E.g., for $\omega = 0.5$ after moving (horizontally) from node `'2-2'` to `'3-2'`, the transition probabilities are 3/8 to `'4-2'` (same direction), 1/8 to `'2-2'` (opposite direction), and 1/4 for `'3-1'` and `'3-3'` (otherwise). Slow–down is similar, exchanging 'same direction' and 'opposite direction'.

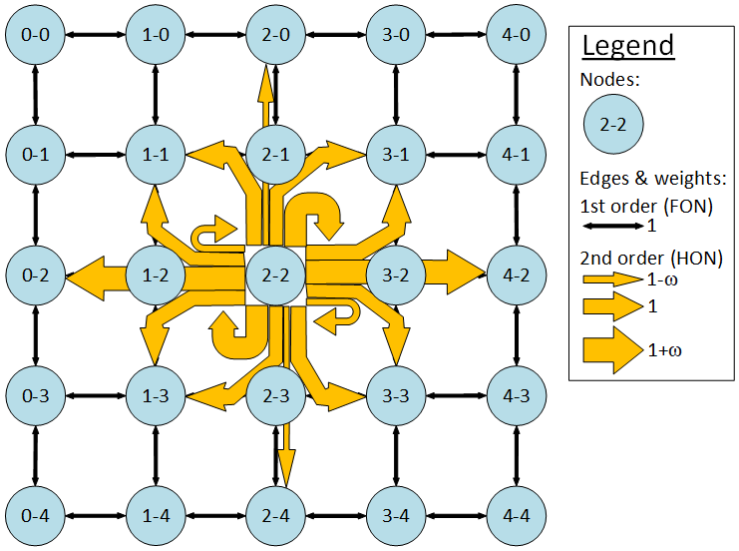


Figure 3.1: *HON Lattice 2D* is a synthetic network extending a first–order two–dimensional lattice. With transition probabilities proportional to weights. We display second–order weights for the node `'2-2'`. For horizontal transitions, moving twice in the same direction is amplified at the expense of U–turns. For vertical transitions, it is the other way round.

The HON lattice 2D comes with several explaining attributes, which do not affect the embeddings but are useful for visualization:

key_len length of the source tuple (1 or 2).

x_orig x–coordinate of the node. If $key\_len = 2$ the last node is used.

x_orig y–coordinate of the node. If $key\_len = 2$ the last node is used.

parity parity ('even' or 'odd') of the sum of the coordinates; see § 2.3.2.

direction 'none' if $key\_len = 1$ and 'left', 'right', 'up', or 'down' otherwise. E.g., the transiton probability $P(N_3 \,|\, N_1 \to N_2)$ has a key (N_1, N_2) with key_length two. The direction of this key is 'left' iff N_2 is the left neighbor of N_1.

There is an interesting symmetry in the walk's distribution, as **a walk has the same probability as its reverse**. Recall that DeepWalk/Node2vec treat walks as bidirectional (due to relying on the implementation of Word2vec), while NetMF does not. Hence, this difference will be masked by that symmetry.

Moreover, the higher-order terms will not affect the (first–order) stationary distribution. For a complete definition (including boundary cases) and proofs, refer to § 3.3.1. We expect HON embeddings of the HON Lattice 2D to exhibit a stretch/compression due to the higher–order dynamic. Measuring this stretch effect is explained in § 3.3.2, and as a by–product, we improve the visualizations in § 3.3.3.

3.3.1 Definition and Properties of the HON Lattice 2D

For a formal definition, we have to establish some vocabulary: Equipping the lattice with cartesian coordinates, we denote the axes as horizontal and vertical. Each node n in the lattice has at most four neighbors, and we write $L(n)$, $R(n)$, $U(n)$, and $D(n)$ to refer to the left, right, upper, and downward neighbors, respectively. To verify valid neighbors, use the following convention:

$$n \text{ has no such neighbor} \iff f(n) = \varnothing, \qquad \forall f \in \{L, R, U, D\}$$

Definition Next, introduce weights, and we will choose the transition probabilities proportional to them below. For the FON, we use an undirected, unweighted lattice; hence each edge has weight one. More formally, we define the first order weights $W^{(1)}_{n \to f(n)}$ as $(\forall f \in \{L, R, U, D\})$:

$$W^{(1)}_{n \to f(n)} = \begin{cases} 0 & f(n) = \varnothing \qquad \text{no such neighbor} \\ 1 & else \end{cases}$$

Since the node degree $d_n = \sum_f W^{(1)}_{n \to f(n)}$, we define $T_{n \to f(n)} := (W^{(1)}_{n \to f(n)})/d_n$.

The second-order weights use a parameter $\omega \in (0, 1)$ to assign modified weights corresponding to speed–up $(1+\omega)$ or slow–down $(1-\omega)$. The goal is to speed–up horizontal and slow–down vertical diffusion. Hence the second–order weights are defined as $(\forall f_1, f_2 \in \{L, R, U, D\})$

$$W^{(2)}_{n \to f_1(n) \to f_2(f_1(n))} = \begin{cases} 0 & if\ f_1(n) = \varnothing \vee f_2(f_1(n)) = \varnothing \qquad \text{no such neighbor} \\ 1 & if\ f_1(f_1(n)) = \varnothing \qquad \text{cannot move twice along } f_1 \\ 1+\omega & if\ (f_1, f_2) \in \{(L,L), (R,R), (U,D), (D,U)\} \qquad \text{speed–up} \\ 1-\omega & if\ (f_1, f_2) \in \{(L,R), (R,L), (U,U), (D,D)\} \qquad \text{slow–down} \\ 1 & else \end{cases}$$

Observe that speed–up and slow–down occur only if $f_1 = f_2$ or $f_2(f_1(n)) = n$ (i.e., f_2 is the inverse of f_1; $f_2 = f_1^{-1}$). To ensure that for fixed n and f_1, choosing both $f_2 := f_1$ and $f_2 := f_1^{-1}$ compensate any weight adjustments in total (i.e., $W^{(2)}_{n \to f_1(n) \to f_1(f_1(n))} + W^{(2)}_{n \to f_1(n) \to f_1^{-1}(f_1(n))} = 2$), we had to exclude the

case where moving twice in the first direction is impossible (i.e., $f_1(f_1(n)) = \varnothing$). This proves that $(\forall f_1)$

$$\sum_{f_2} W^{(2)}_{n \to f_1(n) \to f_2(f_1(n))} = d_{f_1(n)}$$

Therefore define $T^{(2)}_{n, f_1(n)_{1..2} \to f_2(f_1(n))} := (W^{(2)}_{n \to f_1(n) \to f_2(f_1(n))})/d_{f_1(n)}$.

Symmetry Next, we will demonstrate that for any valid random walk (according to FON topology) $N_1 \to \ldots \to N_L$, the **reverse walk** $N_L \to \ldots \to N_1$ has the **same probability** — provided the start node already follows the stationary distribution (i.e., $N_1 \sim \pi$ with $\pi(N_1) = D_{N_1/} \sum_n d_n$). Because all edges have equal weight, all (valid) random walks $N_1 \to N_2$ have the same probability. For $L > 2$, the probability of a random walk is as follows:

$$P(N_1 \to \ldots \to N_L) = \underbrace{\frac{d_{N_1}}{\sum_n d_n} \cdot \frac{W^{(1)}_{N_1 \to N_2}}{d_{N_1}}}_{const} \cdot \prod_{i=2}^{L-1} \frac{W^{(2)}_{N_{i-1} \to N_i \to N_{i+1}}}{d_{N_i}}$$

which is non–zero, because we assume a valid walk. It remains to show that

$$W^{(2)}_{N_{i-1} \to N_i \to N_{i+1}} = W^{(2)}_{N_{i+1} \to N_i \to N_{i-1}}$$

Choose f_1 and f_2 such that $f_1(N_{i-1}) = N_i$ and $f_2(N_i) = N_{i+1}$. Since two conditions result in weight one, we show equality only for weights different from one. The case $f_2 = f_1^{-1}$ is trivial because, for $N_{i-1} = N_{i+1}$, the weights are equal. Then, the only possibility for speed–up or slow–down is $f_1 = f_2$, i.e., moving twice in the same direction, which is invariant under reversing the walk, and $f_1(f_1(N_{i-1})) = N_{i+1} \neq \varnothing$ excludes another exception. This concludes the proof.

Marginal distribution Finally, establish that **second–order transition probabilities do not change the stationary distribution** (if starting with π). Since any walk (of arbitrary length) and its reverse have the same probability, the first and the last node of this walk have identical marginal distribution, thus preserving the stationary distribution.

3.3.2 Measuring the Stretch Effect

Intuitively, the higher–order dynamic affects co-occurrence probabilities in random walks, where speed–up (slow–down) result in higher (lower) probabilities. In § 2.3.2, we noticed that higher probabilities are related to shorter distances in the embedding space. Hence, for a HON embedding based on the skip–gram model and random walks (e.g., HON DeepWalk), we expect the horizontal speed–up and vertical slow–downs to cause a contraction and a stretch in the respective directions of the embedding space.

To measure what we denote as the **stretch effect**, we first define *rightward edges* $\mathcal{E}_R = \{(n, R(n)) \,|\, n \in \mathcal{V} \wedge R(n) \neq \varnothing\}$ and *upward edges* $\mathcal{E}_U = \{(n, U(n)) \,|\, n \in \mathcal{V} \wedge U(n) \neq \varnothing\}$. If the edges' directionality is irrelevant, instead refer to *horizontal and vertical edges* for $\mathcal{E}_R$ and $\mathcal{E}_U$. We calculate the difference

in the embedding space and average its lengths among vertical and horizontal edges. Then we define the *stretch ratio* as:

$$stretch = \frac{\sum_{(u,v)\in\mathcal{E}_U} \|\overrightarrow{u} - \overrightarrow{v}\|/|\mathcal{E}_U|}{\sum_{(u,v)\in\mathcal{E}_R} \|\overrightarrow{u} - \overrightarrow{v}\|/|\mathcal{E}_R|} \tag{3.1}$$

Recall that for LINE–2, the embedding of a lattice consists of two separate clusters, each with common *parity*; see § 2.3.2. The same applies to GraRep, which does not aggregate multi-step transition probabilities over different step sizes. Moreover, LINE–2 is equivalent to DeepWalk, Node2vec, and NetMF with window size $W = 1$ (i.e., a random walk of length one) — which are borderline cases, however. In these cases, the stretch ratio (eq. 3.1) is unsuitable for assessing the stretch effect, and we define the *two-step stretch ratio* as

$$stretch_2 = \frac{\sum_{(u,v)\in\mathcal{E}_{2U}} \|\overrightarrow{u} - \overrightarrow{v}\|/|\mathcal{E}_{2U}|}{\sum_{(u,v)\in\mathcal{E}_{2R}} \|\overrightarrow{u} - \overrightarrow{v}\|/|\mathcal{E}_{2R}|} \tag{3.2}$$

with *rightward two–step edges* $\mathcal{E}_{2R} = \{(n, R(R(n))) \,|\, n \in \mathcal{V} \wedge R(R(n)) \neq \varnothing\}$ and *upward two–step edges* $\mathcal{E}_{2U} = \{(n, U(U(n))) \,|\, n \in \mathcal{V} \wedge U(U(n)) \neq \varnothing\}$.

3.3.3 Visualizing Embeddings of HON Lattice 2D

Alignment We also visualize the embeddings to confirm the stretch effect, though measuring the stretch effect in the embedding space is more meaningful. Also, align the t–SNE visualizations by an *orthogonal transformation* to correctly display the orientation (x–axis: left to right; y–axis: down to up) of a plot. (Both the skip–gram model and t–SNE rely only on scalar products or relative distances, respectively, which are unaffected by orthogonal transformations.) We calculate the *average edge direction* (i.e., differences of node pairs) for $\mathcal{E}_R$ or $\mathcal{E}_U$ in the visualization space to fit the orthogonal transformation. Moreover, we connect points corresponding to adjacent nodes ($\mathcal{E}_R$ and $\mathcal{E}_U$) — or their two-step counterpart ($\mathcal{E}_{2R}$ and $\mathcal{E}_{2U}$) if better suited; see figure 3.2.

Projection Due to the *laborious tuning of t–SNE*, therefore, the *lack of objectivity*, I initially considered using a *projection* instead. This projection assumed that the embedding is contained in a two–dimensional manifold, which is hopefully well–behaved to be approximated by its tangential space. The latter is approximated by the average differences between the embeddings of rightward/upward pairs of nodes.

While the visualizations generated by these projections (figure 3.3) looked interesting, I was finally discouraged when investigating lengths and angles (between individual embedding differences and corresponding averages), as shown in figure 3.4. The individual embedding differences turned out to be close to orthogonal to their corresponding (rightward or upward) averages. Under these circumstances, the projection seems unreliable. Hence, relying on intuition gained in low dimensions when tackling high–dimensional problems can be misleading.

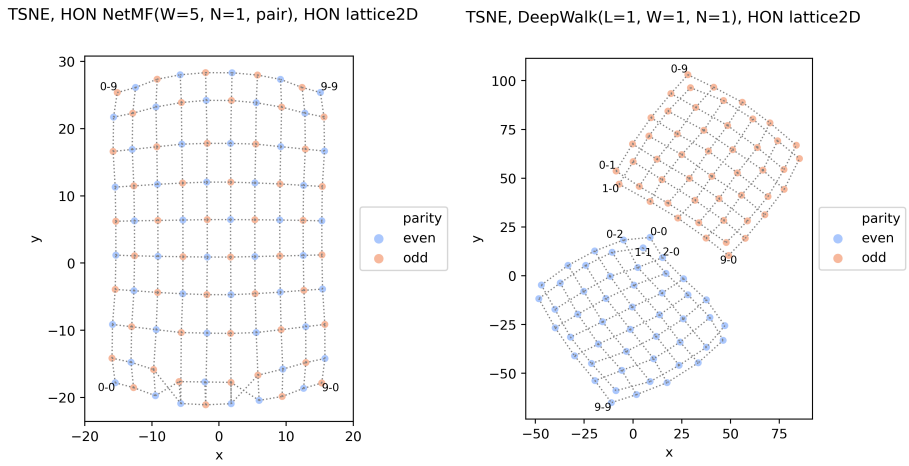


Figure 3.2: Embeddings of the HON Lattice 2D with $\omega = 0.5$. The left plot displays HON NetMF, exhibiting that on average horizontal edges $\mathcal{E}_R$ are shorter than vertical edges $\mathcal{E}_U$ due to the acceleration of horizontal movement. (These edges are both displayed as dotted lines.) The right plot displays a length–one DeepWalk (therefore without stretch effect), and the embedding splits into two clusters, as already seen in figure 2.2 for LINE–2. Hence, the dotted lines display two-step edges ($\mathcal{E}_{2R}$ and $\mathcal{E}_{2U}$) instead. For orientation, color the nodes by parity and annotate some. However, t–SNE reproduces the lattice structure only after tuning.

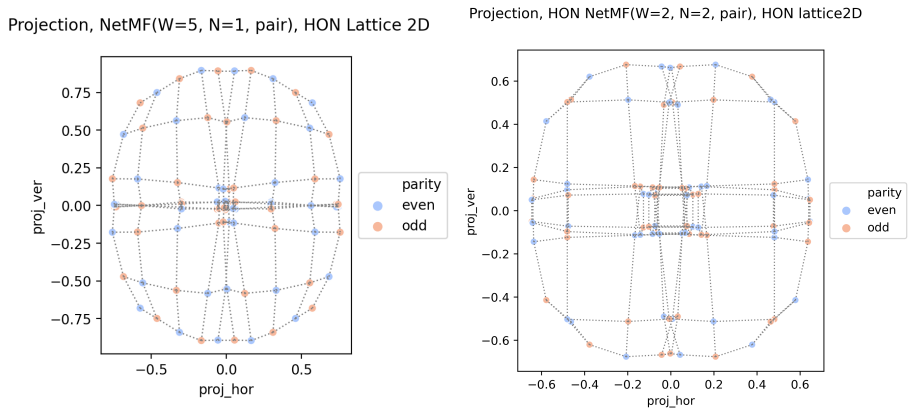


Figure 3.3: Visualization by projection of HON NetMF embeddings of the HON Lattice 2D. The stretch effect results in an asymmetry, but it is hard to tell in which direction for the right one. The same embeddings are visualized by t–SNE in figures 3.2 (left) and 4.2 (top, right).

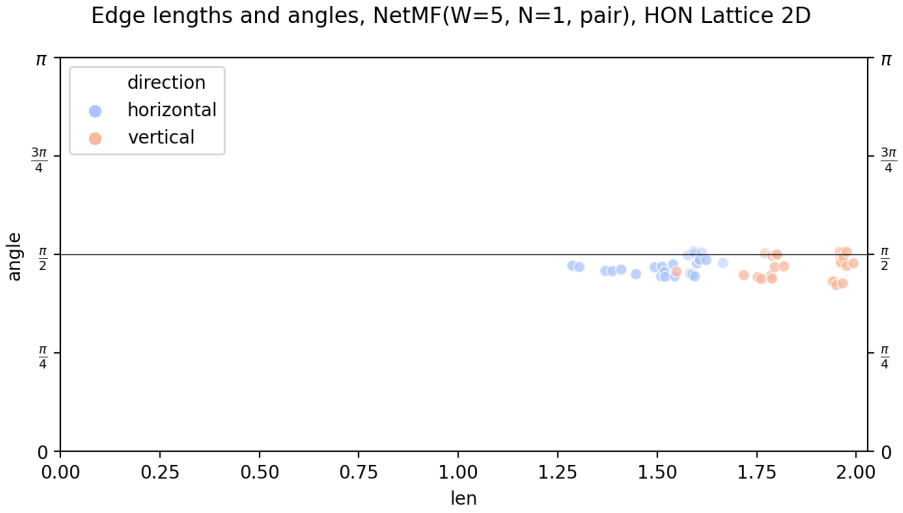


Figure 3.4: Distribution of lengths and angles (towards their average) for horizontal edges $\mathcal{E}_R$ and vertical edges $\mathcal{E}_U$. This nicely illustrates the stretch of the lattice with the average edge lengths for $\mathcal{E}_R$ (1.5) and $\mathcal{E}_U$ (1.8) resulting in a stretch ratio of 1.2. However, the angles between the individual embedding edge–differences and their averages (per direction), indicate that we should not imagine the embedded lattice's manifold as essentially a linear subspace.

Chapter 4

Experimental results

A review of the embeddings mentioned in § 2 reveals the following applications of embeddings ranked by popularity:

1. Classification: HONEM, DeepWalk, Node2vec, NetMF, GraRep, Verse, LINE, InfiniteWalk, GraphSAGE
2. Link prediction: HONEM, Node2vec, Hope, App, Verse, Glee
3. Graph reconstruction: HONEM, Hope, Verse, Glee
4. Visualization: HONEM, GraRep, LINE
5. Clustering: GraRep, Verse
6. Recommendation: Hope, App

Deciding for a **multiclass classification** task is inevitable, allowing to compare all FON baselines against their HON extensions on a terrain of their preference and a real dataset. Furthermore, the **visualization** task is used to preview the classification complexity and explain a synthetic dataset's theoretical properties. We also attempted to predict the probabilities (rather than the existence) of links without success; see § B.1.

4.1 Visualization for a synthetic dataset

In § 3.3, we introduced a two–dimensional lattice with higher–order dynamics. By design, only HON embeddings could notice the difference in diffusion speed between horizontal and vertical movements, resulting in a stretch/compression of the lattice in the embedding space, measured by stretch ratio (3.1) or two–step stretch ratio (3.2) for HON GraRep, and assessed visually.

This experiment will demonstrate that the different HON embeddings considered all exhibit a stretch ratio bigger than one. Importantly, this works for a broad range of parameters, as we are more interested in consistency than determining the 'best' embedding. By comparison, the stretch ratio for FON embeddings deviates from one only due to stochastic variability because FON embeddings do not utilize the higher–order probabilities.

We start with investigating embeddings with 'pair of nodes' interactions before looking at issues specific to embeddings with 'mixed' interactions.

4.1.1 Network

This experiment utilizes the HON Lattice 2D from § 3.3 of size 10×10 with parameter $\omega = 0.5$. For $\omega = 0$, the resulting network is equivalent (w.r.t. the random walks generated) to a FON, which we use for comparison.

The symmetry concerning a reversal in the path's distribution masks differences in directed networks between HON NetMF and HON DeepWalk.

4.1.2 Baselines

HONEM, as well as the HON variants of DeepWalk, Node2vec, NetMF, and GraRep, are all applied to this network, and we evaluate the stretch ratio, see Equations (3.1) and (3.2), on a grid of parameters.

- `dimension` $R \in \{16, 32, 64, 100\}$
- `negative` $N \in \{1, ..., 20\}$
- `window_size` $W \in \{2, ..., 20\}$; for GraRep, the parameter `num_steps` plays a role similar to `window_size`. Note that random walks only utilize second–order probabilities if $W \geq 2$.
- `factor`$=\frac{walk\_length}{window\_size} \in \{1, 2, 4\}$; only for DeepWalk and Node2vec.

To investigate 'mixed' interactions for the other experiment, we use HON NetMF with `pairwise=False` and parameters decided by the 'pairs of nodes' case.

4.1.3 Results

Calculate the stretch Figure 4.1 displays the grid search outcome over the parameters and visualizes it by fixing a parameter and aggregating the others to get the average, minimum, and maximum. For HON DeepWalk and HON GraRep, the lower bounds for the (one– or two–step) stretch ratios are larger than one. For NetMF, this holds for a small number of negative samples N only, as explained below. HONEM has only one parameter, the *dimension* R, and the stretch ratio is larger than one for all with a maximum at $R = 16$ [not shown]. Therefore, we conclude that the stretch effect holds for a wide range of parameters, as predicted theoretically.

Display the stretch Next, we investigate the stretch effect visually and would like to compare HONEM with the other HON and FON embeddings. We will use `dimension` $R = 16$ and `window_size` $W = 2$, as indicated by comparing the concentration around the start node of NetMF's PMI (figure 4.3) and HONEM's neighborhood (figure 4.6, left).

For the HON embeddings, we use their optimal parameters. However, the stretch ratios for FON embeddings deviate from one only due to blind chance, so we apply the corresponding HON parameters instead. Hence, it is fair to use the smallest `window_size` possible ($W = 2$) because figure 4.1 indicates that the stretch ratio increases with W (for small values).

Figure 4.2 displays the t–SNE visualizations of NetMF, GraRep, and DeepWalk for $W = 2$ — each in a FON and a HON variant. Moreover, figure 4.6

(right) displays the t–SNE visualization of HONEM, including the stretch. Note that FON and HON use the same parameters — optimized for HON. These figures confirm that the stretch effect exists for HON while absent for FON.

Mixed case The stretch effect also exists for 'mixed' interactions; see figure 4.5. Moreover, in § 3.2.5, we discussed the *hierarchical structure of walks* and hypothesized about $T^{(k-1)}_{s_{2..k}\to t}$ being a weighted average of $T^{(k)}_{s_{1..k}\to t}$ if stationary (which the HON Lattice 2D is). Despite the transformations by the embedding and t-SNE, the relationships due to the hierarchical structure are remarkedly well represented.

4.1.4 Explanations

Negative Noise contrastive estimation (NCE) — and therefore, negative sampling — compares the probability of a feature to a null–model, the network's (first–order) stationary distribution. Increasing the number of negative samples, therefore, accentuates outstanding features (i.e., $p(c|w)/p_N(c)$ large). For small networks, $p_N(c)$ is already relatively large, requiring a small N. Since matrix factorization operates on the positive $\log(PMI/N)$ values, it amplifies the accentuation of outstanding features even more.

Unlike NetMF, GraRep does not average over different step sizes resulting in a higher concentration of PMI, resulting in an optimal $N = 3$ for HON GraRep compared to $N = 1$ for HON NetMF or HON DeepWalk. (See the upper bounds in figure 4.1; for $W = 2$ in figure 4.2, $N = 2$ is optimal for HON NetMF and HON DeepWalk.)

Window size The parameter `window_size` is the random walk length for NetMF and controls the concentration of the visitation probabilities (and PMI) around the starting node.

Figure 4.3 illustrates the concentration of PMI around a starting point. For small window sizes, the PMI is concentrated around the starting node, and the horizontal speed–up causes the region with high PMI (e.g., > 1) to grow faster horizontally than vertically. However, vertical growth can later catch up since asymptotically $p(c|w)$ converges to the stationary distribution and $PMI \to 1$; see figure 4.4.

The asymmetry of the concentration of PMI around the starting node causes the stretch in the embedding. Combining this explains the stretch ration dependency on the `window_size`; see figures 4.1 and 4.4.

The stretch ratio for HON NetMF seems to prefer even `window_sizes`; see the zig–zag line in the upper–right of figure 4.1. In § 2.3.2, we already noticed that a pair of adjacent nodes have opposite parity. Hence, the nodes visited in a walk (excluding the start node) have balanced parity for even walk lengths and unbalanced for odd ones. Using the two–step stretch ratio (3.2) instead, which is suitable for embeddings splitting into two groups (e.g., GraRep), confirms that parity caused the zig–zag line [not shown].

Node2vec The simulation of Node2vec included bias parameters $p = 1/q \in \{.5, 1, 2, 4\}$. For $p = 2$ and $p = 4$, diffusion is boosted in all directions, resulting

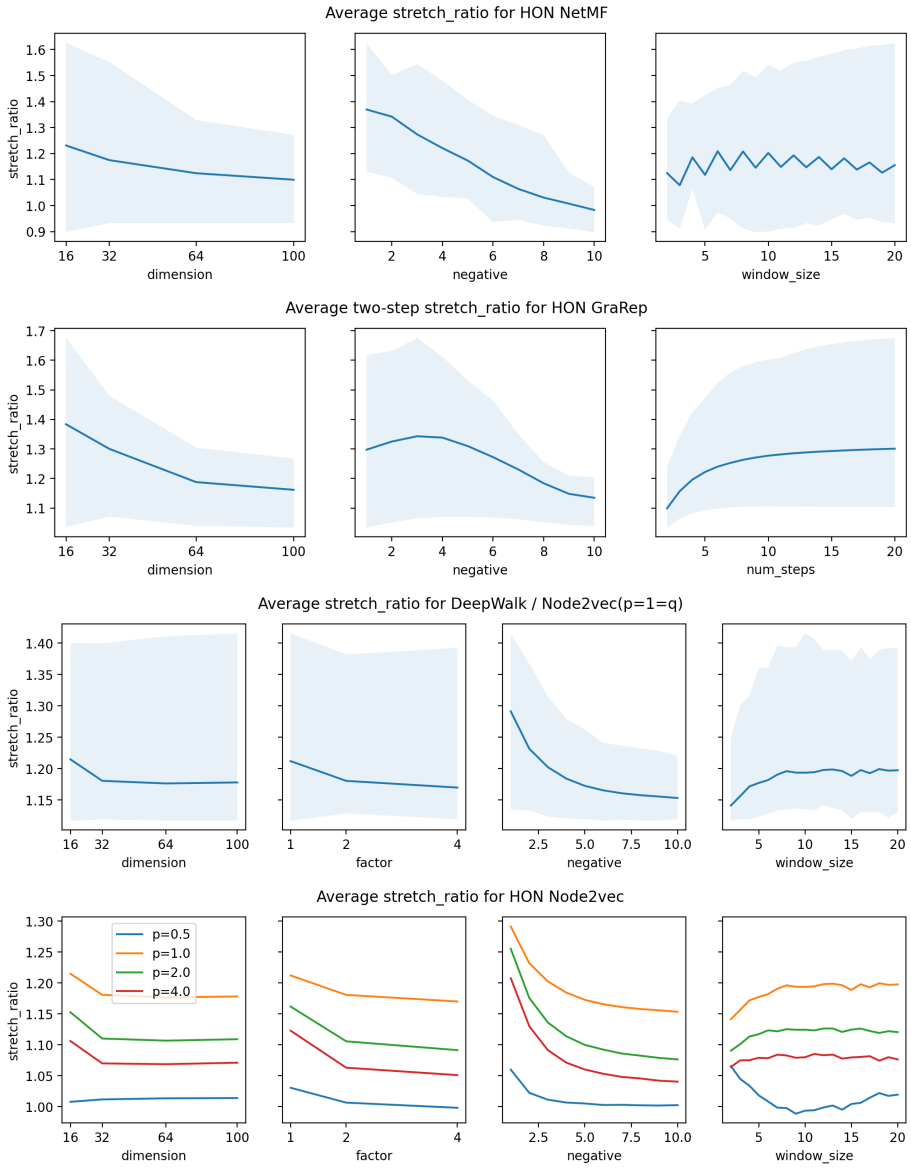


Figure 4.1: The stretch ratio for embeddings of HON Lattice 2D is calculated on a grid of parameters. Note that the minimal `window_size` or `num_steps` must be two to utilize the second–order dynamics. For visualization purposes, we fix one parameter and aggregate over the others. The individual plots display the mean as well as the range (except for Node2vec). Using a small number of negative samples, N seems crucial to get a large stretch ratio. Moreover, the biased random walks of HON Node2vec do not increase the stretch ratio compared to DeepWalk.

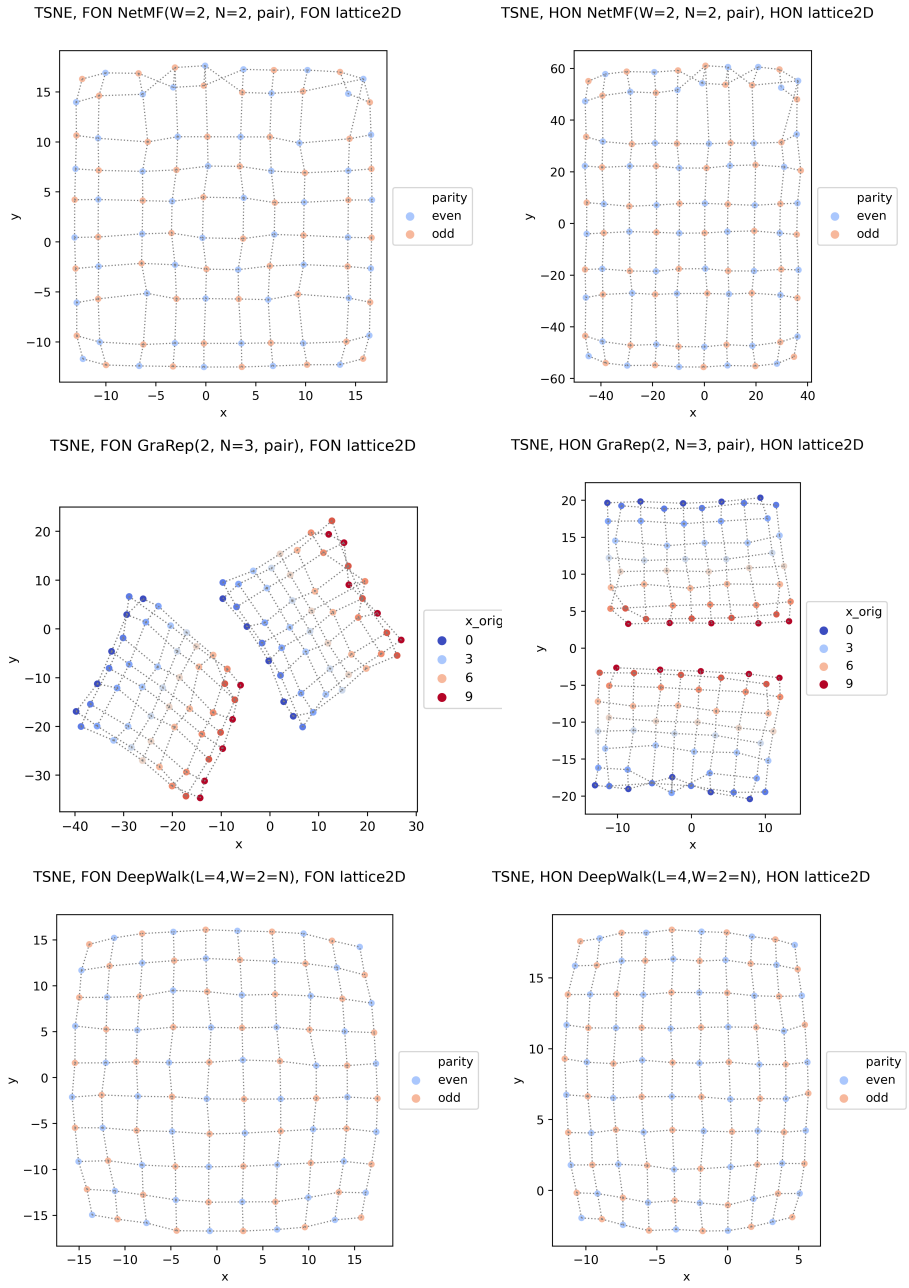


Figure 4.2: Compare of FON vs. HON embeddings. For the FON embeddings, the stretch ratio is one. The HON embeddings have stretch ratios of 1.34 (NetMF), 1.24 (GraRep), or 1.15 (DeepWalk). The `window_size` ($W = 2$) was chosen for its similarity to HONEM (with a stretch ratio of 1.33, see figure 4.6). According to figure 4.1, the stretch effect is more pronounced for larger `window_sizes`. Recall that the stretch ratio is directly calculated on the embedding, while the t–SNE visualization required some manual parameter tuning.

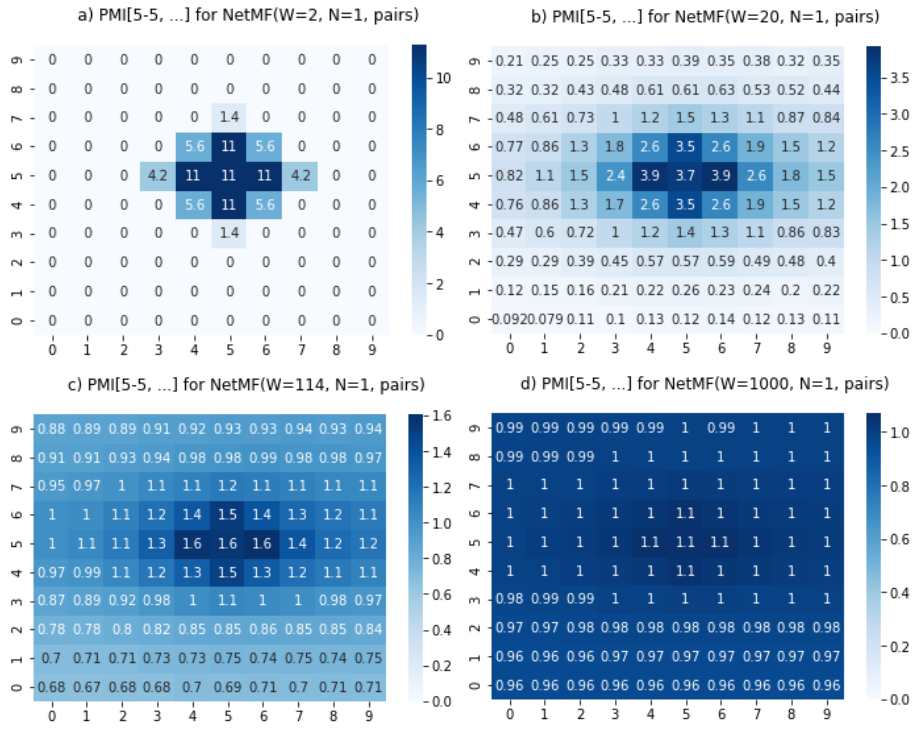


Figure 4.3: Display the concentration around the starting node '5-5' of the pointwise mutual information (PMI) for different `window_sizes` $W \in \{2, 20, 114, 1000\}$ of (pairwise) HON NetMF with one negative sample (N). HON NetMF factorizes $\log(PMI/N)^+$, and PMI is defined as the random walk visitation probability divided by the stationary distribution. The second–order dynamic of the HON Lattice 2D ($size = 10 \times 10, \omega = 0.5$) was constructed to speed–up horizontal and slow–down vertical transitions. Consequently, horizontal neighbors' embeddings are, on average, closer than vertical ones which is measured by the stretch ratio, Eq. (3.1). This effect attains its maximum for window size $W = 110$ and vanishes asymptotically for $PMI \to 1$.

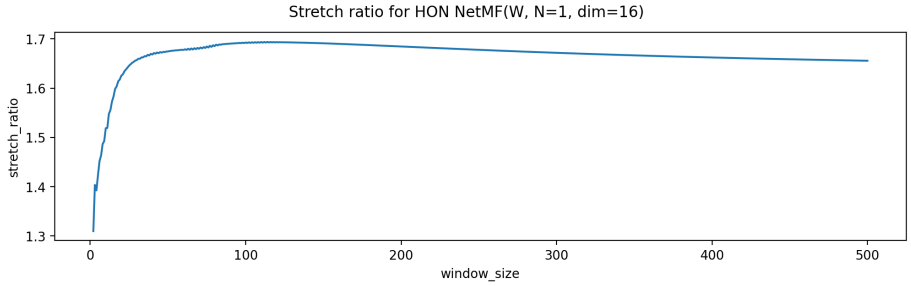


Figure 4.4: HON NetMF: stretch ratios for varying window sizes. The maximum 1.69 is at $W = 114$.

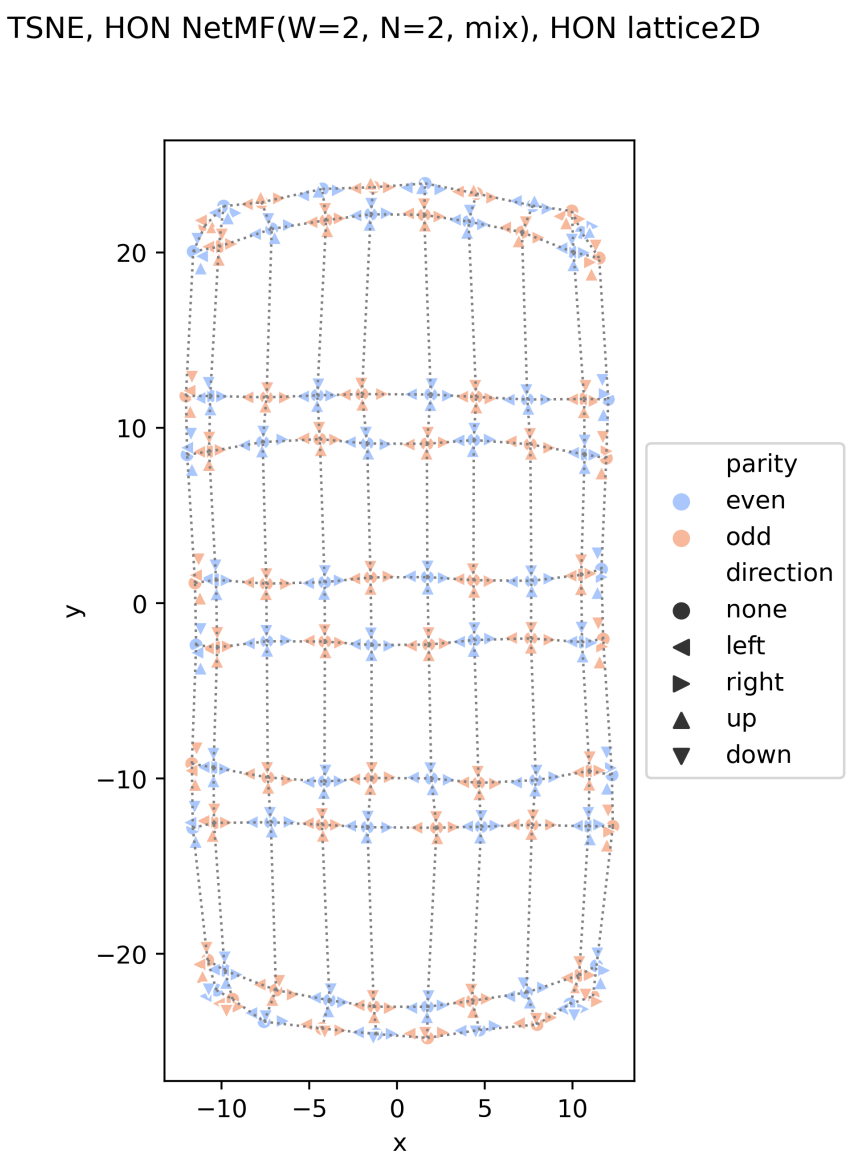


Figure 4.5: A HON NetMF embedding with 'mixed interactions' visualized by t–SNE displays the stretch effect (stretch ratio 1.23) — the grouping in pairs of rows disappears for higher dimensions. Several tuples (to be embedded) *end* with the same node and therefore have identical neighborhoods and similar embeddings. Since horizontal transitions are speeded up, a start tuple of nodes corresponding to a rightward movement (▶) has a higher probability of moving rightward again and is therefore embedded towards its right neighbor. Conversely, starting with an upward movement (▲) increases the probability of moving downwards, resulting in an embedding closer to its downward neighbor. For leftward (◀) and downward (▼) movement, similar patterns emerge. Finally, starting with a single node (●) results in a mix of the above.

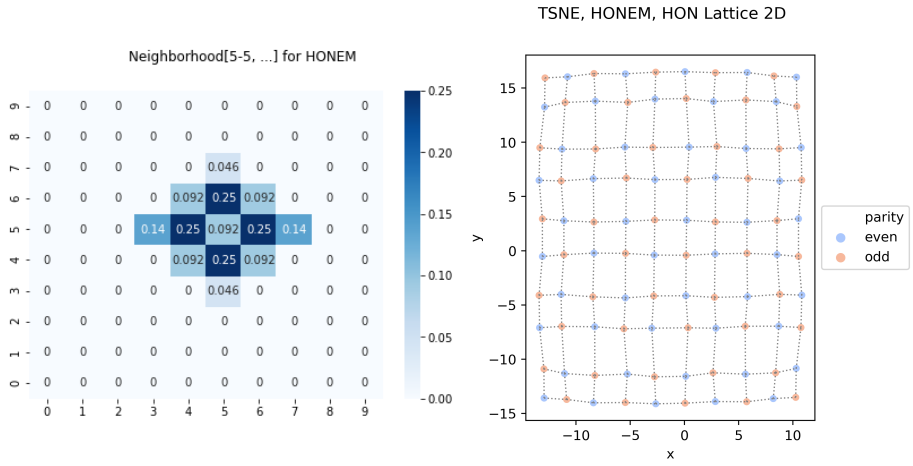


Figure 4.6: HONEM: neighborhood for the node '5-5' and t–SNE visualization of the embedding (with dimension 16). The stretch ratio is 1.33.

in lower stretching compared to $p = 1$ (i.e., DeepWalk) For $p = 0.5$, the stretching effect no longer appears consistently for different parameters. Hence, this specific application does not benefit from the additional tuning parameters for the bias of Node2vec compared to DeepWalk.

Factor For DeepWalk, the word–context pairs are generated from a random walk of length `factor*window_size`. While NetMF generally approximates DeepWalk, there are minor differences: first, nodes near the ends of a walk contribute to fewer word–context pairs, and second, the distribution of the words $p(w)$ varies from almost uniform (small `walk_length`) to stationary (large `walk_length`); see also [25, Remark 1].

I decided to allow `walk_length` different from `window_size` in the simulation but will not investigate this further, as the HON Lattice 2D is too well behaved to exploit these differences.

4.2 Multiclass Classification for SocioPatterns

Apply the network embeddings to a real dataset for multi–class classification (see § 4.2.1). We will focus on comparing HON embeddings against their FON equivalents. Hence, use default settings for the classifiers and watch for a consensus instead of perfectly fitting a single classifier.

4.2.1 Methodology

Multi–class classification aims to predict the labels for the nodes of a network from its embedding.

In practice, some nodes are labeled, and we have to estimate the labels for the others. However, for a simulation study, all nodes are labeled, and we split the nodes into training and test sets to predict the labels of the nodes in the test

set based on the ones in the training set. Note that we still use all the nodes to calculate the embedding.

In k–fold cross–validation, we randomly split the nodes into $k = 5$ sets $\mathcal{V}_1, ...\mathcal{V}_k$ of roughly equal size and repeatedly ($i = 1 \dots k$) estimate the labels for the test set of nodes $\mathcal{V}_i$ based on the other nodes ($\mathcal{V} - \mathcal{V}_i$) as the training set. Eventually, we have collected the predicted label and the probabilities for each label, from which we could calculate the *confusion matrix*, the *F1–scores*, or the *receiver operating characteristic* (ROC) curve. The confusion matrix counts the number of combinations of true and predicted label pairs.

The F1–score for binary classification is the *harmonic mean of precision and recall*, $(\text{F1–score})^{-1} = \frac{1}{2}(precision^{-1} + recall^{-1})$, with values between zero and one (bigger is better). Recall (a.k.a. true–positive rate, TPR) is a conditional probability of predicted given the true label. The precision is similar, with true and predicted labels exchanged. For multi–class classification, the difference between F1–micro and F1–macro is, whether we aggregate before or after the harmonic mean.

The ROC curve graphically depicts a fundamental trade–off in hypothesis testing between type 1 and type 2 errors. It uses the vocabulary **true–positive rate (TPR)** and **false–positive rate (FPR)**, which correspond to $1 - P(type\ 1\ error)$ and $P(type\ 2\ error)$, respectively. The area under the ROC curve describes the toughness of a classification task. Generalizing from binary to multi–class classification, we can calculate the ROC curve for one class against either another class (one vs. one) or any other class (one vs. rest) and decided for the latter.

Moreover, we may repeat the k–fold cross–validation to reduce variability (due to random split) or get error bounds.

4.2.2 Networks

For the classification task, we need datasets with labels and a HON structure. SocioPatterns.org provides temporal network data, and I decided to use the following three ones.

Primary school temporal network data [7, 31] The labels name teachers or classes (for children).

Contacts in a workplace [11] The labels contain departments.

Hospital ward dynamic contact network [36] The labels identify patients and different kinds of staff.

All these record interactions with a 20 seconds resolution. Hence the next step is extracting[1] causal paths (named 'walks' in this thesis). There is a tuning parameter Δ for the path extraction, which specifies the maximal time–difference allowed for two temporal edges to be included in a causal path.

Next, we fit a multi–order network (MON, [29]) and estimate the higher–order Markov model's optimal order K. Finally, export the resulting models using the same data format as BuildHON+ [39] for later use in our software.

[1] Similar to task 1.4 of `https://ingoscholtes.github.io/kdd2018-tutorial/`

The maximal Delta reflects my computational resources, and I will continue with the following[2]:

- PrimarySchool, $\Delta = 20$ sec, $K = 2$
- Workplace, $\Delta \in \{1 \text{ min}, 5 \text{ min}, 10 \text{ min}, 20 \text{ min}, 30 \text{ min}\}$, $K = 2$
- HospitalWard, $\Delta \in \{20 \text{ sec}, 40 \text{ sec}, 1 \text{ min}\}$, $K = 3$ (or 2 for Δ=20 sec)

4.2.3 Baselines

Given the task of classifying nodes from their embeddings, we consider only HON embeddings from § 3.2 with 'pairs of nodes' interactions. (There is simply no use for embeddings of tuples of nodes longer than one.)

Each of the following embeddings consists of a HON and a FON variant, distinguished by their respective prefixes. Both variants use the same implementation, but the the FON variant is applied to a network converted from HON to FON (by discarding higher-order probabilities).

DeepWalk generates 100 walks per node, each of length $L = 80$. The `window_size` is $W = 10$; Optimization uses negative sampling with one iteration.

GraRep(S=4) generates 4 embeddings for the different step sizes and concatenates them together. Therefore, each individual embedding uses only dimension 16, i.e., a fourth of the dimension.

NetMF(W=10) generates an embedding with `window_size` $W = 10$.

NetMF(W=2) generates an embedding with `window_size` $W = 2$ because both PrimarySchool, Workplace, and HospitalWard ($\Delta = 20$") have maximal order $K = 2$.

Node2vec(p=0.5) uses the bias parameters $p = 0.5, q = 2$ and generates the embedding exactly as DeepWalk.

Node2vec(p=2) uses the bias parameters $p = 2, q = 0.5$ and generates the embedding exactly as DeepWalk.

HONEM HONEM has neither parameters (besides the `dimension`) nor a pair of FON/HON variants.

Each embedding has `dimension`[3] $R = 64$, uses one negative sample ($N = 1$), and uses only a 'pair of nodes' interaction.

The simulation uses the following classifiers with default settings (and for reproducibility `random_state=1`):

LR Logistic regression (`sklearn.linear_model.LogisticRegression`)

RF Random forest (`sklearn.ensemble.RandomForestClassifier`)

SVC Support vector classification (`sklearn.svm.SVC`)

[2]Due to the larger size than a dataset from a similar domain, I considered both 'Highschool dynamic' and 'Workplace 2nd' as a lower priority. Other datasets were too complicated or lacked metadata.

[3]I compared dimensions 32 and 64 earlier without noticing differences and did not reassess my decision since. Instead, I later focused on comparing different Δ values.

	f1-macro			f1-micro		
	FON	HON	HONEM	FON	HON	HONEM
Hospital 20"	(20) 52%	(17) 53%	(38) 32%	(20) 61%	(18) 63%	(37) 47%
Hospital 40"	(18) 46%	(19) 45%	(37) 30%	(18) 55%	(21) 52%	(29) 48%
Hospital 1'	(16) 47%	(21) 42%	(35) 31%	(19) 55%	(21) 53%	(23) 50%
PrimarySchool	(26) 77%	(11) 83%	(38) 41%	(26) 82%	(11) 87%	(38) 45%
Workplace 1'	(14) 77%	(23) 73%	(38) 57%	(15) 88%	(22) 87%	(38) 74%
Workplace 5'	(18) 74%	(19) 73%	(38) 56%	(17) 86%	(20) 85%	(38) 72%
Workplace 10'	(14) 78%	(23) 73%	(38) 55%	(18) 85%	(19) 85%	(38) 73%
Workplace 20'	(15) 79%	(22) 74%	(38) 52%	(21) 85%	(16) 86%	(38) 68%
Workplace 30'	(10) 85%	(27) 73%	(38) 55%	(11) 90%	(26) 87%	(38) 73%

Table 4.1: Comparing *average* F1–scores (and their ranks in parentheses) among different embeddings and classifiers, with the embeddings grouped by FON, HON, and HONEM. For each combination of a dataset, Δ, and type of score (micro or macro), we underline the values of the best group, as well as the best Δ per dataset. An undisputed winner exists for PrimarySchool and Workplace (Δ=30'), while Hospital[Ward] (Δ=20") is close and also has lower F1-scores.

4.2.4 Results

We calculate the F1–scores (macro and micro) of every combination of embedding and classifier, display them as a scatter plot in figure 4.7, and provide a summary in table 4.1; see appendix A for the raw results.

PrimarySchool For Δ=20 sec (the only one available for this dataset), HON embeddings are superior to their FON counterparts, and an average rank of 11 among 18 items is exceptional — no FON embedding made it into the top ten. Moreover, the two F1–scores are highly correlated. HONEM performs worst by a huge margin.

Workplace For Δ=30 min, the FON embeddings outperform their HON counterparts in an equally (or even slightly more) resounding success as the other way around for PrimarySchool.

Things get closer for other Δ values, and the two F1–scores agree less, with FON still winning for F1–macro and a tie between FON and HON for F1–micro. HONEM performs worst by a large margin.

HospitalWard The lower F1–scores and larger discrepancies between them indicate that this dataset is harder to classify. FON wins for Δ=40 sec and 1 min, but for Δ=20 sec, HON achieves the highest average F1–scores (among all Δ values).

HONEM performs still worst for the F1–macro score (though without margin) and better for the F1-micro score. Surprisingly, while both HONEM and our HON embeddings utilize the higher–order terms, HONEM performs best when the FON embeddings have an advantage over the HON ones. (However, this is only anecdotal evidence.)

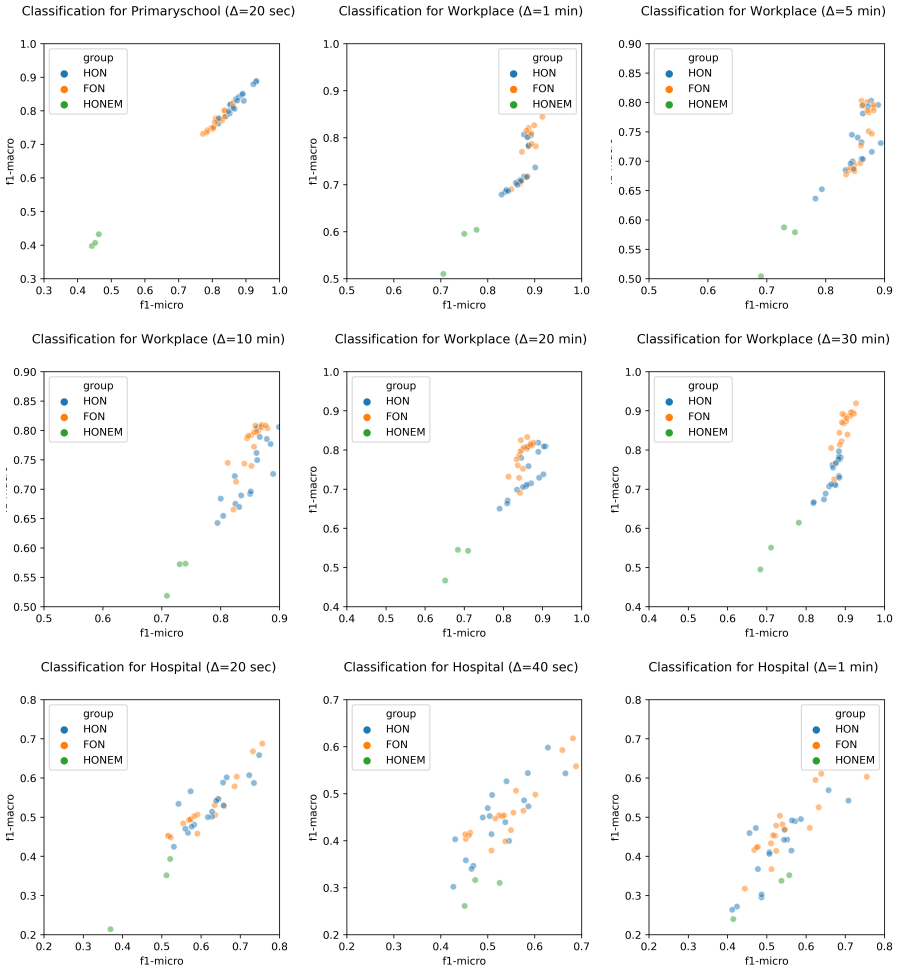


Figure 4.7: F1–scores (macro and micro) for pairs of embeddings and classifiers; see also table 4.1.

Best embeddings Next, we compare the classifications of a FON vs. a HON embedding for each dataset. For each, we display a t–SNE visualization of the embedding, a confusion matrix, and (one–vs–rest) ROC curves — both calculated from 10 repetitions. Hence, Figures 4.8, 4.9, and 4.10 display the best combinations of FON/HON embedding and classifier for PrimarySchool (Δ=20 sec), Workplace (Δ=30 min), and HospitalWard (Δ=20 sec), respectively; where 'best' means the highest sum of F1–micro and F1–macro.

4.2.5 Explanations

Rosvall et al. [27] describe the effect of using HON on spreading dynamics and community detection and mention algorithms to investigate related phenomena.

Modularity Comparing Figures 4.8 and 4.10, we notice the classes' cluster structure in the PrimarySchool network's embeddings, which is not the case for

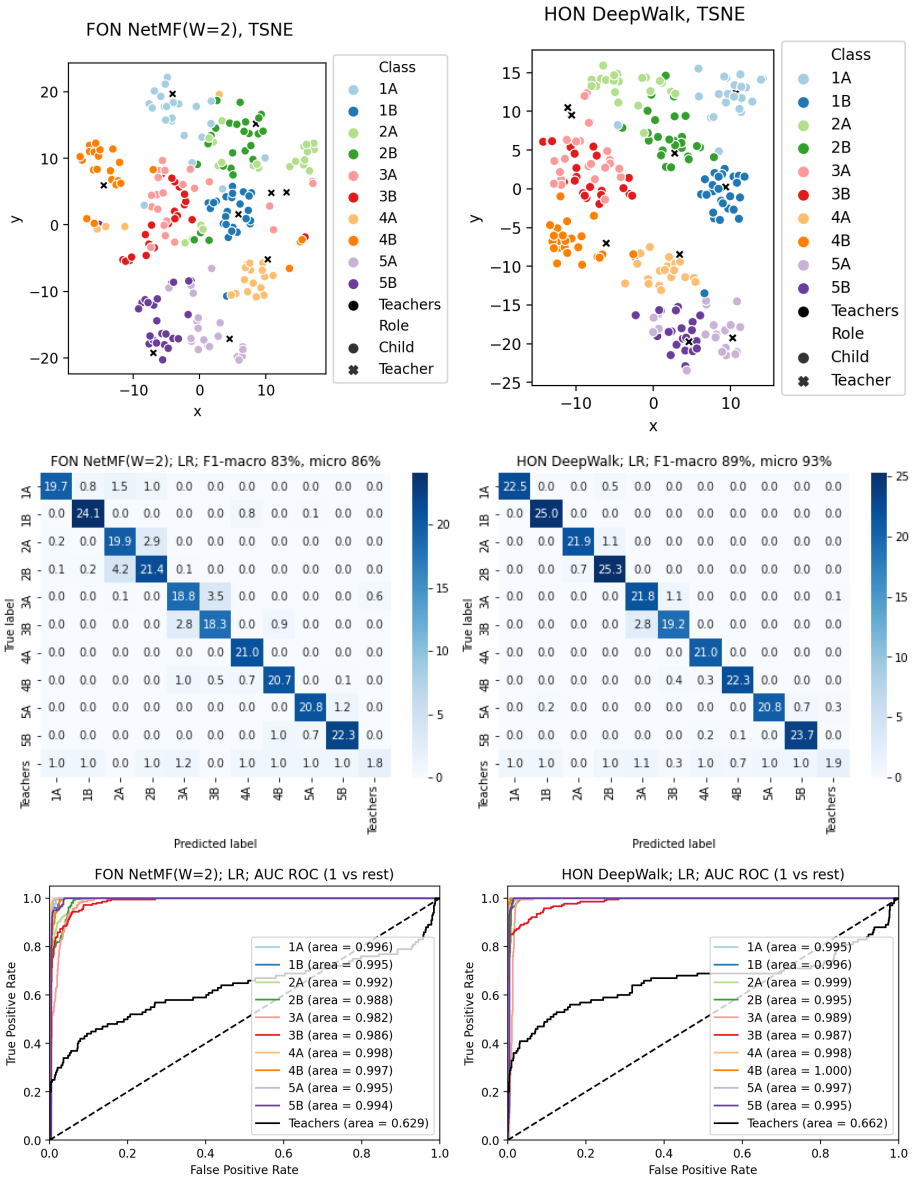


Figure 4.8: For the PrimarySchool network, combine several classifiers and embeddings and select the best pair (w.r.t. F1–micro + F1–macro) containing a FON (left) or a HON (right) embedding, respectively. These were 'FON NetMF(W=2)' and 'HON DeepWalk', both combined with logistic regression. T–SNE visualization (top) displays the clustering of children into classes, which is more pronounced for HON, confirmed by the confusion matrix (center). Both embeddings are unsuitable for classifying teachers. The area under ROC (bottom) illustrates the separability of any class from the rest.

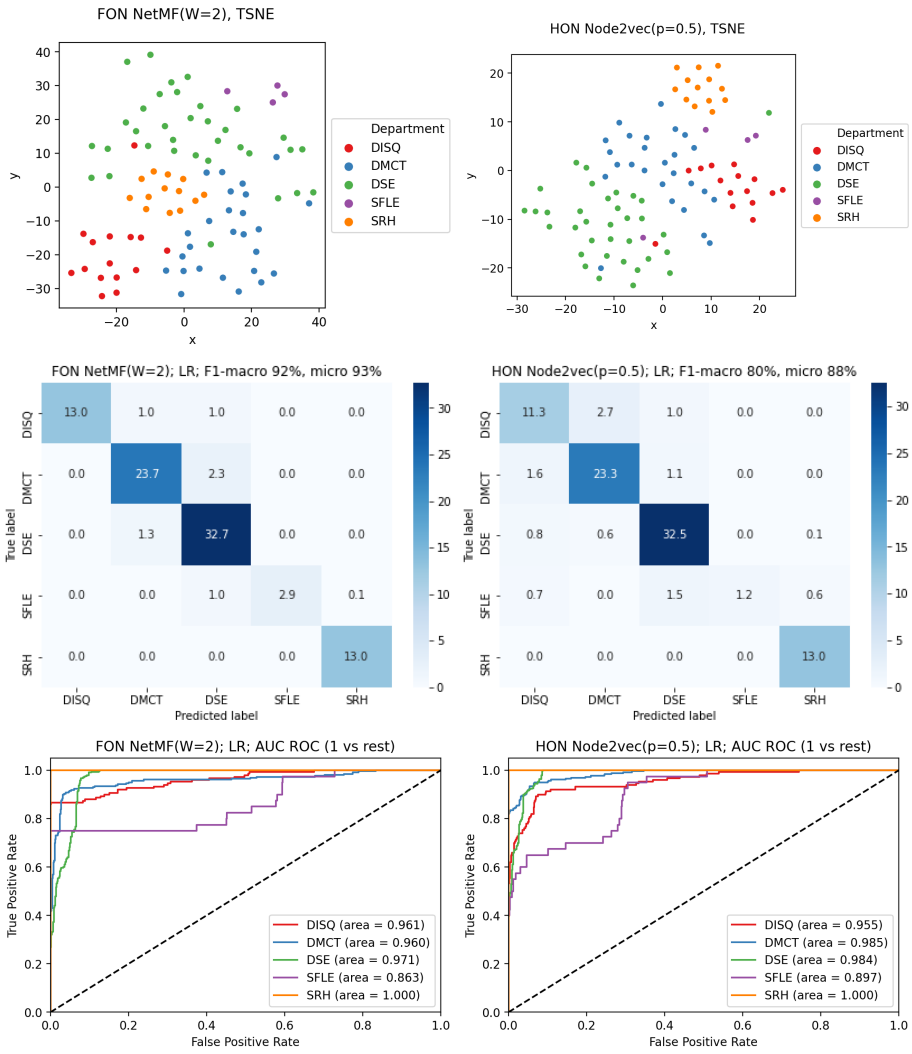


Figure 4.9: For the Workplace network with Δ=30 min, combine several classifiers and embeddings and select the best pair (w.r.t. F1–micro + F1–macro) containing a FON (left) or a HON (right) embedding, respectively. These were 'FON NetMF(W=2)' and 'HON Node2vec(p=0.5)', both combined with logistic regression. T–SNE visualization (top) displays the clustering of individuals into departments. It is hard to decide visually, which one separates the clusters better, but the confusion matrix (center) reveals the differences, with FON performing better. The area under ROC (bottom) illustrates the separability of any class from the rest.

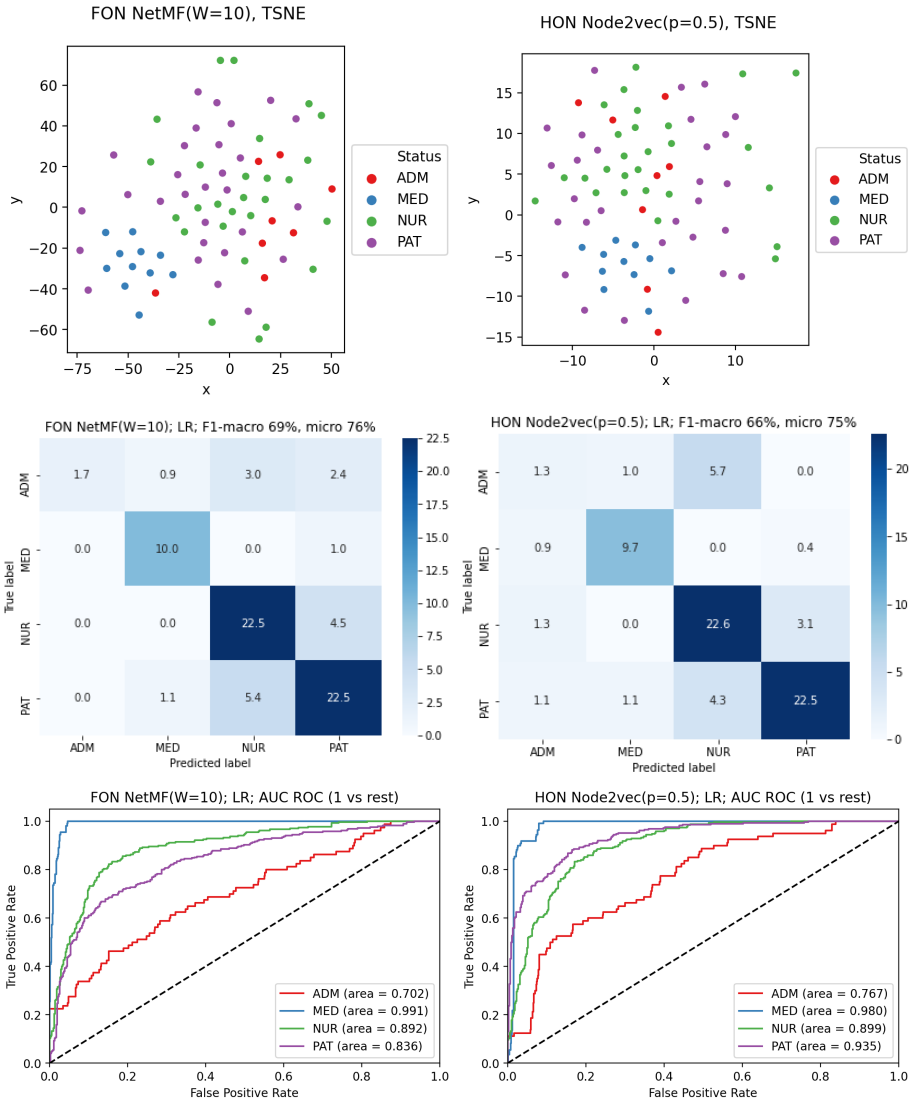


Figure 4.10: For the HospitalWard network with Δ=20 sec, combine several classifiers and embeddings and select the best pair (w.r.t. F1–micro + F1–macro) containing a FON (left) or a HON (right) embedding, respectively. These were 'FON NetMF(W=10)' and 'HON Node2vec(p=0.5)', both combined with logistic regression. T–SNE visualization (top) displays the absence of clustering. FON performs slightly better than HON, see the confusion matrix (center), despite HON embeddings performing slightly better on average. The area under ROC (bottom) illustrates the separability of any class from the rest.

HospitalWard.

Literature defines the modularity for unweighted and undirected networks. The generalization to weighted [22] and directed [18, Eq. (3)] is straight forward.

Using higher–order De Bruijn graphs allows us to calculate the modularity for higher orders. Note that the adjacency matrix becomes increasingly sparse for higher dimensions, and it is essential to avoid iterating over all pairs of nodes[4].

The modularities are 44% (PrimarySchool), 47% (Workplace), and 25% (HospitalWard), independent of the order. This investigation confirms that HospitalWard is more challenging to classify.

Temporal aspects for PrimarySchool Why do HON embeddings consistently outperform their FON counterparts for PrimarySchool? It turns out that the classification task performs poorly for teachers since they interact primarily with children rather than their peers. Examining the visualizations reveals that some teachers belong to a specific class (they are teaching), while others are separate. Hence, a teacher interacting with children of more than one class may function in a FON as a communication hub between children from different classes, which is impossible in HON, as the interaction takes place separately in time. Figure 4.11 confirms this kind of mechanism, although it is infrequent.

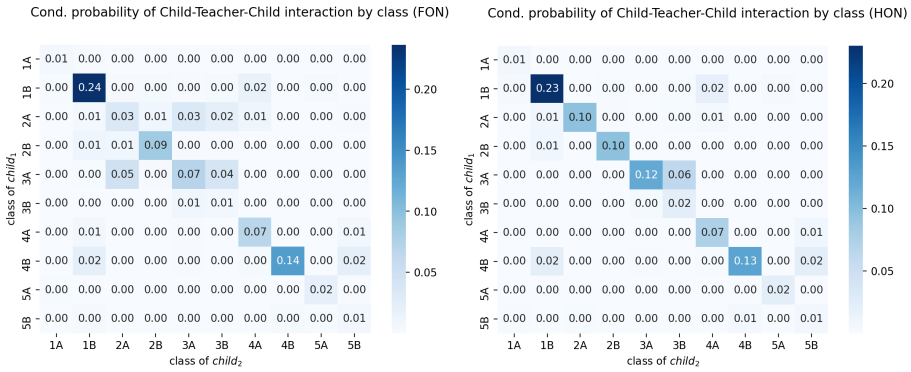


Figure 4.11: Examining interaction sequences from child to teacher and then to another child. FON ignores the temporal aspect when linking two children via the teacher, creating faulty connections between children of different classes sharing the same teacher. Observing that the diagonal contains more probability mass for HON than for FON confirms the hypothesis. The heatmaps display probabilities of classes of a pair of children. The children's pair follows the marginal distribution of random walks with the pattern $child_1 \to teacher \to child_2$ (generated by rejection sampling from the stationary distribution). However, these walks are rare ($< 0.4\%$).

Entropy of Transition Probabilities In § 3.2.5, we described the hierarchical structure of the start–tuples of a walk and demonstrated that $T^{(k-1)}_{s_{2..k} \to t}$ is a weighted average of $T^{(k)}_{s_{1..k} \to t}$ if the transition probabilities were *stationary*.

[4]Hint: $\sum_{i,j} k_i^{in} k_j^{out} \delta_{c_i,c_j} = \sum_c (\sum_i k_i^{in} \delta_{c_i,c}) \cdot (\sum_j k_j^{out} \delta_{c_j,c})$ in [18, Eq. (3)]

Consequently, $T^{(k-1)}_{s_{2..k}\to t}$ would tend to be less concentrated than $T^{(k)}_{s_{1..k}\to t}$, which we measure by their respective entropies.

Figure 4.12 confirms that higher-order probabilities are more concentrated (less entropy) than lower ones; see also [27, table 2]. This is in line with the previous observation that HON transitions are more restrictive than FON ones.

Examining transition probabilities' entropies in figure 4.12 also reveals that the PrimarySchool has higher average entropy than Workplace (Δ=30 min). Is the Workplace (Δ=30 min) insufficiently connected for a HON embedding? Unfortunately, I did not notice a pattern when comparing the Workplace network's entropies for different Δ values [not shown].

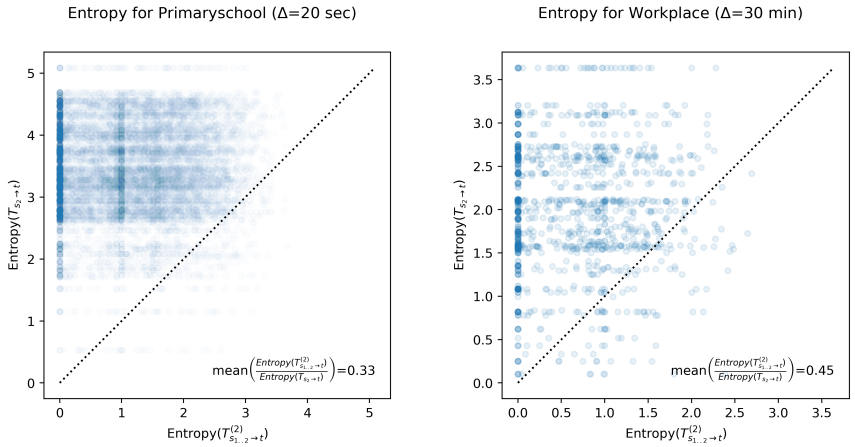


Figure 4.12: Comparing the entropy of pairs of transition probabilities $T^{(2)}_{s_{1..2}\to t}$ and $T_{s_2\to t}$ for PrimarySchool (Δ=20 sec, left) and Workplace (Δ=30 min, right). The second–order transitions' entropy for *both networks* is considerably *smaller* than the first–order ones (on average). Note that PrimarySchool has higher average entropy than Workplace: 3.5 vs. 2.3 (first–order) and 1.1 vs. 0.7 (second–order), respectively.

Chapter 5

Conclusion and Discussions

5.1 Summary and Reflection

Representation learning for first-order networks (FON) is already an established research area that produced various embedding methods, which provide great starting points for higher-order network (HON) adaptations. For clarification, HON refers to a network with random walks following a higher–order Markov model.

An important early assumption, which shaped this thesis was that 'pairwise interactions are not enough'; see § 3.1.2. This originates from objective two, which demanded generating paths from a known embedding. There is no way to decode higher–order transition probabilities from pairwise interactions (e.g., $\langle \overrightarrow{w}, \overrightarrow{c} \rangle$ in the skip–gram model with nodes w and c) without losing information. Hence any approximation of $T^{(k)}_{s_{1..k} \to s_{k+1}}$ must formally depend on all nodes of the tuple $s_{1..k+1}$.

In the end, it turned out to be sufficient to replace random walks by their HON counterparts (see § 3.1.1), and there was a lack of applications for the advanced approaches.

After setting the direction initially and analyzing the problem, we had plenty of options, a luxury which complicated decisions. The thesis reflects this by explaining the choices made. We have to finish with time running out and summarize some highlights, although there are still more interesting questions to explore.

HON Lattice 2D This synthetic network is useful for understanding the skip–gram model and explaining the impact of second–order dynamics. When implementing an embedding, it is valuable for debugging to have a theoretical understanding of the expected outcome when embedding a lattice.

Classification To publish a new method, one has to demonstrate the new method's usefulness by surpassing existing methods in a comparison experiment. Fortunately, the HON embeddings outperform their FON counterparts for the PrimarySchool dataset, although the result for the Workplace dataset spoils the good impression.

However, the fundamental differences between FON and HON random walks are underlying circumstances determining whether HON or FON embeddings are a better tool. Ideally, we could determine the differences resulting from choosing a HON or FON random walk and assess them by subject–specific criteria.

Probability Prediction This experiment was the only opportunity to evaluate the approaches based on the above mentioned 'pairwise is not enough' assumption, and is therefore mentioned despite its failure. Moreover, it raises an important question: should an embedding generalize or reproduce a network?

HONEM Theoretical considerations reveal weaknesses of HONEM, but we acknowledge that fixing these weaknesses would result in an entirely different embedding, which is computationally more expensive. In our classification experiment, HONEM performed poorly.

5.2 Threats to Validity and Limitations

Implications of using HON We noticed in § 4.2.5 that HON might prohibit walks, which are valid in FON. This may remove unwanted connections between groups or break up clusters. Hence, we cannot conclude that HON embeddings are better than FON ones — even if the HON differs significantly from the FON, which is therefore incorrect. Criteria to assist this decision would be useful.

Stationary Distribution For a FON random walk, setting the start distribution $P_{\overline{S_1}}$ equal to the stationary distribution π is sufficient for a stationary random walk. Therefore, it is natural for NetMF to set $p_N = \pi$.

The maximal order transition probabilities ($T^{(K)}_{s_{1\ldots K}\to t}$) solely determine the stationary distribution of HON random walks. For a stationary distribution of the walk, we need to choose all other probabilities (start distribution and transition probabilities of order k smaller than the maximal order K) accordingly; see § 2.1.3, estimating probabilities. Should we somehow ignore the lower order probabilities when generating random walks? Furthermore, should we replace the FON stationary distribution by a marginal of the HON one, given the maximal order terms are the hardest to estimate?

No big issue; I have just decided to stick with the FON stationary distribution without considering alternatives.

5.3 Future Work

Fast SGD–based Implementations For a thesis, matrix factorization based embeddings are rewarding, as we do not need to worry about convergence. My experimental SGD implementation (§ 3.2.5) lacks performance, putting any thoughts to include HON App/Verse in the evaluations to an end; see also [26] about tuning gensim's Word2vec for Python. However, for large datasets, fast SGD implementations are essential. A HON App/Verse implementation is currently missing, and HON DeepWalk/Node2vec relies on Word2vec (which treats word and context symmetrically).

Applications for Embeddings Beyond Pairs of Nodes Is there any real–world application for embeddings with 'mixed' (or 'pairs of walks') interactions? We only attempted probability prediction (§ B.1) without success.

Appendix A

Results for the Classification task

The following tables contain the detailed results corresponding to figure 4.7 and table 4.1

	f1-macro			f1-micro		
	LR	RF	SVC	LR	RF	SVC
FON DeepWalk	0.772	0.742	0.749	0.815	0.786	0.797
FON GraRep(S=4)	0.777	0.771	0.746	0.824	0.829	0.801
FON NetMF(W=10)	0.799	0.768	0.777	0.836	0.817	0.810
FON NetMF(W=2)	0.825	0.813	0.801	0.863	0.862	0.837
FON Node2vec(p=0.5)	0.782	0.751	0.748	0.837	0.803	0.803
FON Node2vec(p=2)	0.766	0.732	0.736	0.808	0.772	0.783
HON DeepWalk	0.889	0.799	0.848	0.931	0.848	0.888
HON GraRep(S=4)	0.784	0.792	0.762	0.841	0.852	0.818
HON NetMF(W=10)	0.835	0.805	0.818	0.870	0.867	0.852
HON NetMF(W=2)	0.841	0.830	0.819	0.879	0.894	0.855
HON Node2vec(p=0.5)	0.887	0.808	0.851	0.930	0.863	0.890
HON Node2vec(p=2)	0.879	0.778	0.831	0.922	0.819	0.874
HONEM	0.397	0.407	0.432	0.442	0.452	0.463

Table A.1: F1–scores for the PrimarySchool dataset. Note that the best performing non–HON methods are 'FON NetMF(W=2)' with logistic regression and F1–macro 0.825 and 'FON GraRep(S=4)' with random forest and F1–micro 0.873, both have rank 12. HONEM performs worst by large margin.

	f1-macro			f1-micro		
	LR	RF	SVC	LR	RF	SVC
FON DeepWalk	0.782	0.809	0.770	0.902	0.886	0.873
FON GraRep(S=4)	0.718	0.707	0.718	0.884	0.872	0.885
FON NetMF(W=10)	0.781	0.702	0.705	0.887	0.868	0.867
FON NetMF(W=2)	0.803	0.690	0.705	0.885	0.850	0.862
FON Node2vec(p=0.5)	0.845	0.821	0.826	0.916	0.887	0.899
FON Node2vec(p=2)	0.787	0.815	0.809	0.893	0.883	0.892
HON DeepWalk	0.806	0.686	0.716	0.892	0.843	0.883
HON GraRep(S=4)	0.717	0.706	0.710	0.877	0.867	0.868
HON NetMF(W=10)	0.782	0.686	0.684	0.887	0.838	0.837
HON NetMF(W=2)	0.784	0.689	0.679	0.887	0.840	0.829
HON Node2vec(p=0.5)	0.807	0.703	0.737	0.877	0.860	0.901
HON Node2vec(p=2)	0.801	0.700	0.708	0.885	0.863	0.871
HONEM	0.510	0.604	0.595	0.705	0.776	0.750

Table A.2: F1–scores for the Workplace dataset with Δ=1 min. FON performs best. HONEM performs worst by large margin.

	f1-macro			f1-micro		
	LR	RF	SVC	LR	RF	SVC
FON DeepWalk	0.747	0.803	0.792	0.878	0.861	0.880
FON GraRep(S=4)	0.677	0.697	0.686	0.835	0.859	0.848
FON NetMF(W=10)	0.751	0.684	0.683	0.873	0.839	0.849
FON NetMF(W=2)	0.783	0.689	0.694	0.873	0.841	0.849
FON Node2vec(p=0.5)	0.786	0.801	0.796	0.882	0.871	0.882
FON Node2vec(p=2)	0.727	0.795	0.786	0.860	0.863	0.871
HON DeepWalk	0.803	0.740	0.716	0.877	0.854	0.878
HON GraRep(S=4)	0.696	0.704	0.687	0.842	0.861	0.847
HON NetMF(W=10)	0.781	0.700	0.652	0.863	0.846	0.793
HON NetMF(W=2)	0.745	0.685	0.636	0.845	0.834	0.783
HON Node2vec(p=0.5)	0.796	0.732	0.731	0.889	0.861	0.893
HON Node2vec(p=2)	0.793	0.795	0.703	0.872	0.862	0.863
HONEM	0.504	0.579	0.587	0.690	0.748	0.729

Table A.3: F1–scores for the Workplace dataset with Δ=5 min. FON performs best. HONEM performs worst by large margin.

	f1-macro			f1-micro		
	LR	RF	SVC	LR	RF	SVC
FON DeepWalk	0.797	0.787	0.806	0.861	0.845	0.867
FON GraRep(S=4)	0.713	0.772	0.665	0.826	0.857	0.822
FON NetMF(W=10)	0.800	0.745	0.792	0.864	0.812	0.852
FON NetMF(W=2)	0.797	0.744	0.740	0.857	0.840	0.852
FON Node2vec(p=0.5)	0.805	0.808	0.809	0.870	0.859	0.870
FON Node2vec(p=2)	0.804	0.791	0.808	0.879	0.847	0.876
HON DeepWalk	0.785	0.762	0.692	0.878	0.861	0.850
HON GraRep(S=4)	0.675	0.689	0.670	0.825	0.835	0.832
HON NetMF(W=10)	0.806	0.722	0.655	0.863	0.824	0.804
HON NetMF(W=2)	0.802	0.684	0.642	0.861	0.800	0.795
HON Node2vec(p=0.5)	0.806	0.789	0.726	0.899	0.866	0.889
HON Node2vec(p=2)	0.777	0.749	0.696	0.885	0.862	0.851
HONEM	0.519	0.573	0.572	0.709	0.740	0.730

Table A.4: F1–scores for the Workplace dataset with Δ=10 min. FON performs better for F1–macro while FON and HON are on par for F1–micro. HONEM performs worst by large margin.

	f1-macro			f1-micro		
	LR	RF	SVC	LR	RF	SVC
FON DeepWalk	0.811	0.786	0.818	0.871	0.840	0.877
FON GraRep(S=4)	0.732	0.752	0.691	0.813	0.850	0.842
FON NetMF(W=10)	0.801	0.817	0.803	0.848	0.870	0.861
FON NetMF(W=2)	0.833	0.761	0.729	0.861	0.837	0.840
FON Node2vec(p=0.5)	0.808	0.796	0.814	0.862	0.843	0.870
FON Node2vec(p=2)	0.776	0.825	0.806	0.834	0.845	0.852
HON DeepWalk	0.819	0.780	0.711	0.889	0.846	0.860
HON GraRep(S=4)	0.708	0.729	0.715	0.858	0.891	0.871
HON NetMF(W=10)	0.813	0.664	0.671	0.874	0.810	0.811
HON NetMF(W=2)	0.808	0.699	0.650	0.861	0.835	0.790
HON Node2vec(p=0.5)	0.809	0.795	0.738	0.908	0.889	0.902
HON Node2vec(p=2)	0.809	0.759	0.705	0.903	0.864	0.850
HONEM	0.467	0.543	0.545	0.651	0.710	0.684

Table A.5: F1–scores for the Workplace dataset with Δ=20 min. FON performs better for F1–macro while FON and HON are on par for F1–micro. HONEM performs worst by large margin.

	f1-macro			f1-micro		
	LR	RF	SVC	LR	RF	SVC
FON DeepWalk	0.896	0.873	0.887	0.914	0.902	0.913
FON GraRep(S=4)	0.805	0.768	0.725	0.864	0.876	0.872
FON NetMF(W=10)	0.897	0.839	0.893	0.918	0.905	0.922
FON NetMF(W=2)	0.919	0.813	0.822	0.927	0.886	0.890
FON Node2vec(p=0.5)	0.888	0.868	0.881	0.899	0.896	0.903
FON Node2vec(p=2)	0.844	0.892	0.871	0.885	0.892	0.891
HON DeepWalk	0.782	0.783	0.734	0.888	0.884	0.884
HON GraRep(S=4)	0.712	0.710	0.689	0.872	0.875	0.850
HON NetMF(W=10)	0.776	0.664	0.707	0.886	0.818	0.859
HON NetMF(W=2)	0.762	0.674	0.667	0.867	0.846	0.820
HON Node2vec(p=0.5)	0.797	0.755	0.730	0.884	0.868	0.885
HON Node2vec(p=2)	0.777	0.767	0.712	0.883	0.876	0.864
HONEM	0.495	0.614	0.551	0.684	0.782	0.711

Table A.6: F1–scores for the Workplace dataset with Δ=30 min. FON performs best. HONEM performs worst by large margin.

	f1-macro			f1-micro		
	LR	RF	SVC	LR	RF	SVC
FON DeepWalk	0.453	0.506	0.494	0.519	0.591	0.572
FON GraRep(S=4)	0.603	0.579	0.548	0.691	0.685	0.643
FON NetMF(W=10)	0.688	0.528	0.531	0.756	0.659	0.635
FON NetMF(W=2)	0.668	0.458	0.506	0.732	0.591	0.636
FON Node2vec(p=0.5)	0.452	0.494	0.492	0.516	0.573	0.569
FON Node2vec(p=2)	0.448	0.502	0.484	0.523	0.583	0.555
HON DeepWalk	0.588	0.546	0.530	0.656	0.644	0.657
HON GraRep(S=4)	0.602	0.607	0.471	0.665	0.723	0.560
HON NetMF(W=10)	0.566	0.476	0.481	0.573	0.576	0.583
HON NetMF(W=2)	0.534	0.424	0.460	0.543	0.531	0.567
HON Node2vec(p=0.5)	0.658	0.501	0.587	0.748	0.628	0.735
HON Node2vec(p=2)	0.542	0.500	0.514	0.639	0.617	0.628
HONEM	0.214	0.351	0.393	0.369	0.512	0.521

Table A.7: F1–scores for the HospitalWard dataset with Δ=20 sec. HON performs best. The scores are lower compared to the other datasets, indicating that classification does not work well on this dataset.

	f1-macro			f1-micro		
	LR	RF	SVC	LR	RF	SVC
FON DeepWalk	0.452	0.506	0.454	0.531	0.560	0.535
FON GraRep(S=4)	0.498	0.558	0.460	0.601	0.688	0.555
FON NetMF(W=10)	0.618	0.423	0.464	0.681	0.549	0.576
FON NetMF(W=2)	0.593	0.379	0.399	0.659	0.508	0.537
FON Node2vec(p=0.5)	0.454	0.417	0.447	0.524	0.463	0.516
FON Node2vec(p=2)	0.404	0.414	0.411	0.453	0.452	0.460
HON DeepWalk	0.526	0.452	0.439	0.540	0.504	0.537
HON GraRep(S=4)	0.486	0.543	0.414	0.577	0.665	0.508
HON NetMF(W=10)	0.449	0.400	0.347	0.489	0.545	0.469
HON NetMF(W=2)	0.403	0.358	0.302	0.431	0.453	0.427
HON Node2vec(p=0.5)	0.598	0.544	0.473	0.628	0.585	0.587
HON Node2vec(p=2)	0.497	0.469	0.340	0.509	0.500	0.465
HONEM	0.261	0.316	0.310	0.451	0.473	0.525

Table A.8: F1–scores for the HospitalWard dataset with Δ=40 sec. FON performs best. The scores are lower compared to the other datasets, indicating that classification does not work well on this dataset.

	f1-macro			f1-micro		
	LR	RF	SVC	LR	RF	SVC
FON DeepWalk	0.424	0.482	0.453	0.477	0.540	0.521
FON GraRep(S=4)	0.525	0.603	0.473	0.632	0.755	0.609
FON NetMF(W=10)	0.611	0.433	0.414	0.639	0.511	0.524
FON NetMF(W=2)	0.595	0.318	0.367	0.624	0.444	0.512
FON Node2vec(p=0.5)	0.454	0.479	0.469	0.516	0.524	0.545
FON Node2vec(p=2)	0.416	0.503	0.423	0.468	0.533	0.475
HON DeepWalk	0.490	0.411	0.443	0.572	0.507	0.552
HON GraRep(S=4)	0.569	0.542	0.415	0.657	0.708	0.563
HON NetMF(W=10)	0.459	0.295	0.272	0.456	0.487	0.424
HON NetMF(W=2)	0.472	0.303	0.264	0.472	0.487	0.412
HON Node2vec(p=0.5)	0.467	0.367	0.406	0.545	0.477	0.507
HON Node2vec(p=2)	0.495	0.492	0.442	0.587	0.564	0.544
HONEM	0.240	0.352	0.338	0.415	0.557	0.537

Table A.9: F1–scores for the HospitalWard dataset with Δ=1 min. FON performs best. The scores are lower compared to the other datasets, indicating that classification does not work well on this dataset.

Appendix B

Additional Experimental results

The following experiments are relegated to the appendix because the results are less relevant.

B.1 Probability Prediction

According to the review in § 4, the second–most popular experiment is link prediction. The point about link prediction is assuming an incompletely observed network and guessing (unknown!) edges from the embedding. Experiments mimic this with cross–validation.

However, the existence of a link depends only on the network topology, a first–order concept. Moreover, 'mixed' interaction embeddings embed the start-tuple $s_{1..k}$ of $T^{(k)}_{s_{1..k}\to t}$, which already reveals a few valid edges.

Hence, let us predict transition probabilities $T^{(k)}_{s_{1..k}\to t}$ instead. Conveniently, we need an embedding with 'mixed' interactions for $k > 1$, which we had no opportunity to evaluate yet.

We predict the transition probabilities from the embedding by modifying formula (2.4) to consider the network topology:

$$\hat{p}(t|s_{1..k}) = \begin{cases} \frac{\exp\left(\langle \overrightarrow{s_{1..k}}, \overrightarrow{t} \rangle\right)}{const(s_{1..k})} & if\ (s_k, t) \in \mathcal{E} \\ 0 & else \end{cases}$$

This corresponds to the `decode`–method mentioned in § 3.2 with the parameters `use_neighborhood=True` and `normalize=True`.

B.1.1 Network

A network, whose probabilities an embedding manages to reproduce perfectly, is ill–suited because generalization is critical when dealing with incomplete data. The network's structure should be resilient to removing or adding a few edges. A power grid or the internet looks promising due to the redundancy.

However, this is a topic for future research, and we simply use the PrimarySchool network from § 4.2.2.

B.1.2 Baselines

The approaching deadline for submitting the thesis allows only for a small simulation study. We compare HON and FON variants of NetMF with the following parameters:

- `dimension` $R \in \{16, 32, 64, 128\}$; smaller values increase the approximation error
- `negative` $N \in \{1, 5\}$; larger values reduce the accuracy of the approximation
- `window_size` $W \in \{1, 2, 3, 5\}$; larger values blur the probabilities

B.1.3 Best Case: Complete Information

Let us check first if probability prediction works at all, i.e., with complete information about the network.

NetMF(W=1, N=1, R=128) and NetMF(W=5, N=5, R=16)[1] represent two opposite poles: The former aims for good prediction, while the latter aims for stability.

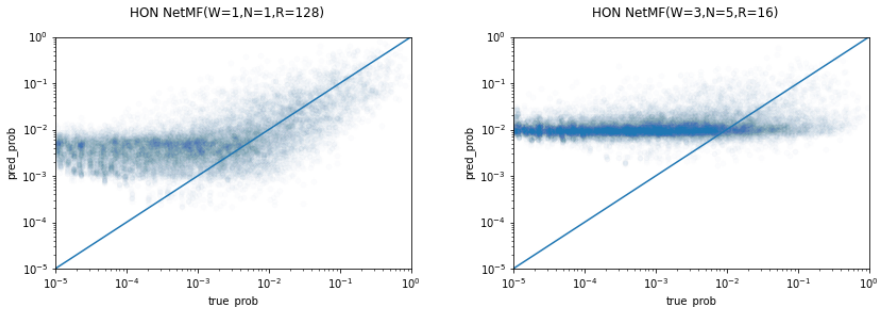


Figure B.1: As expected, HON NetMF(W=1, N=1, R=128) is better at predicting probabilities than NetMF(W=3, N=5, R=16)

B.1.4 Methodology

The goal of cross–validation is to verify transition probabilities encoded by the embedding while making sure this information has not been used to train the embedding.

Cross-validation for Weighted Optimization Comparing matrix factorization via SVD and SGD, we notice that the latter is a weighted optimization problem while the former is an unweighted one. Setting the weight to zero is an easy way to encode missing information in a weighted optimization problem, which does not work for the unweighted one. Consequently, this approach works only for embeddings based on SGD.

Instead of splitting the data into training and test, we select a set of pair of nodes (FON edges) $\mathcal{X}$ and make sure the `update(u, v, ...)` step in learning

[1] We use NetMF(W=3, N=5, R=16) in the figures instead

the embeddings is skipped if the pair $(w, c) \in \mathcal{X}$. If w or c are higher–order De Bruijn nodes, we use the pair `(w[-1], w[-1])` instead, which consists of the last FON nodes of w and c only. (This should also apply to the negative samples.) This effectively means replacing $p(w, c)$ by the corresponding conditional distribution on $(w, c) \notin \mathcal{X}$.

Finally, compare those pairs excluded by $\mathcal{X}$ the transition probabilities with their counterparts derived from the embedding and measure the approximation's quality by a quadratic loss. This approach has the disadvantage that we need control of the SGD optimization, which is complicated when using an existing implementation such as gensim's Word2vec. (Not to mention that SGD needs more fine–tuning than SVD.)

Non–informative transition probabilities Another idea is to tweak the probabilities, removing the information about the original value in order to use matrix factorization. Let s define $\mathcal{X}_w = \{c|(w, c) \in X\}$ ('pair of nodes' case) or $\mathcal{X}_w = \{c|(w[-1], c) \in X\}$ ('mixed' case).

Because higher–order transition probabilities already involve walks implies that the FON edges are already known. In order to hide any information about $p(c|w)$ for c in $\mathcal{X}_w$, we replace it with the uniform distribution ($1/d_w^{out}$, where d_w^{out} is the outdegree of w). This us our definition of *non–informative*.

$$\tilde{p}(c|w) = \begin{cases} 1/d_w^{out} & if\ c \in X_w \\ p(c|w) \cdot \frac{1-|X_w|/d_w^{out}}{1-\sum_{x \in X_w} p(x|w)} & else \end{cases}$$

This is basically the idea for this experient: Use cross–validation to generate $\mathcal{X}$, create an embedding based on $\tilde{p}(c|w)$, and compare true and predicted probabilities.

B.1.5 Results

Using the SSE criterion (sum of squared errors between true and predicted probability) — see the end of § C.11 — those estimators perform best, which perform worst in the best case (figure B.1). Unfortunately, this is not just a trade–off between good fit and the ability to generalize — it is just a shrinkage effect: a constant is a better approximation of the true probabilities than an uncorrelated prediction.

B.2 HON Penalized Transition Embedding

We never displayed the experimental estimator (§ 3.2.5), because we did not find an applications for 'mixed' interactions, and figure 4.5 uses HON NetMF. Let us remedy this omission, see figure B.3.

Visualizing the loss during training (figure B.4), we observe a ledge, which is because the two parts (positive and negative) of the loss ($\mathcal{L}(w, c)$, see § 3.2.5) counteract each other:

$$\mathcal{L}(w, c) := -\big(\underbrace{p(w, c) \log \sigma(\langle \overrightarrow{w}, \overrightarrow{c} \rangle)}_{positive} + \underbrace{N \cdot p(w) \cdot p_N(c) \log \sigma(-\langle \overrightarrow{w}, \overrightarrow{c} \rangle)}_{negative} \big)$$

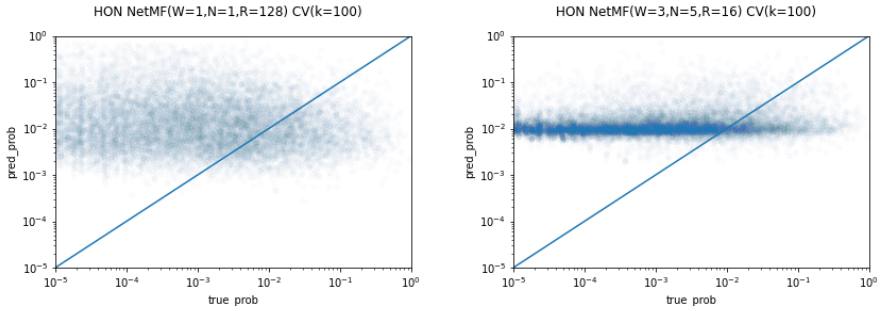


Figure B.2: Comparing true vs. predicted probabilities using cross–validation, using an non–informative distribution. Unfortunately, the probability prediction breaks down when using a non–informative distribution with 1% of the FON edges excluded.

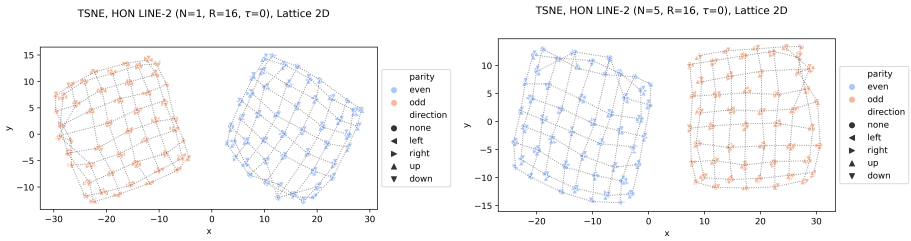


Figure B.3: Comparing the experimental estimator without penalty (i.e., HON LINE–2) for one (left) and five (right) negative samples N. The result is as expected when considering figures 4.5 and 2.2 (right).

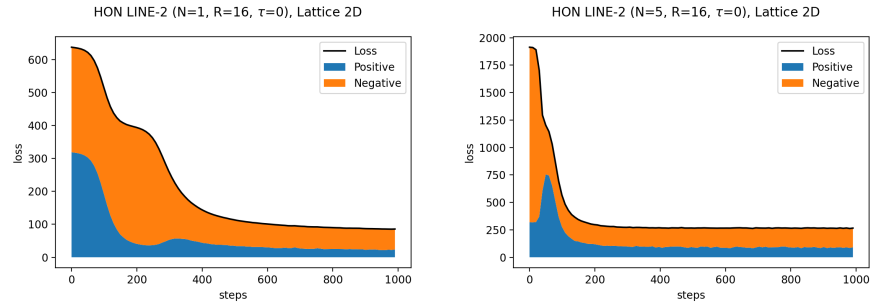


Figure B.4: Splitting the loss during training of the experimental estimator into positive and negative part explains the ledge in the sum. Note that for few negative samples ($N = 1$) the positive part is optimized first, while for larger ones ($N = 5$) it is the other way round.

Appendix C

Source Code

The codebase is divided between infrastructure and applications (e.g., code for an experiment or a specific figure). For the former, we provide information about its usage and display the source code in full length (§ C.2 to § C.6). For the latter, we display only the jupyter notebooks.

There is also a change in terminology: throughout the thesis, I avoided the term 'path' and used 'walk' instead because is is unclear whether a path must have distinct edges[1]. However, I noticed this late and did not refactor the code afterward.

C.1 Introduction and Usage

HigherOrderPathGenerator First, we have to create an instance of the class `HigherOrderPathGenerator` and load the rules, configuration, and metadata; see `Plots_ExpClass.ipynb`. The rules are the transition probabilities are stored in a dictionary (rules$[s_{1..k}]$=$\{t \quad : \quad T^{(k)}_{s_{1..k}\to t}\}$). The metadata contains information about the nodes, used for visualization and classification. The configuration identifies the rules (or transition probabilities) and will be exported together with the figure in `Visualizations.save_describe()`; see below.

```
from HigherOrderPathGenerator import HigherOrderPathGenerator
# read data/models/primaryschool_1.config into dictionary model_config
with open('data/models/primaryschool_1.config', 'r') as f:
  model_config = { items[0]:'␣'.join(items[1:]) for items in [
                   line.strip('\n').split('\t') for line in f.readlines() ] }
gen = HigherOrderPathGenerator(node_sort_key=int, id='primaryschool_1',
    config=model_config)
gen.load_BuildHON_rules('data/models/primaryschool_1.csv')
# read data/models/metadata_primaryschool.csv into dictionary metadata
with open('data/models/metadata_primaryschool.csv', 'r') as f:
  metadata = {int(i):Ci for i,Ci [line.split() for line in f]}
gen.add_metadata('Class', metadata, use_last=True)
gen.add_metadata('Role', { i:'Teacher' if c=='Teachers' else 'Child' for i,c
    in metadata.items()})
# checks
print(list(gen.check_transition_probs(tol=1e-14)))
print(list(gen.verify_stationarity(1, tol=1e-15)))
```

To avoid duplicate code, we may also use:

[1] https://en.wikipedia.org/wiki/Path_(graph_theory)

```
from Datasets import init_generator
gen = init_generator('primaryschool_1.csv')
```

Alternatively, a HigherOrderPathGenerator corresponding to the HON Lattice 2D (§ 3.3) is instantiated with:

```
from SyntheticNetworks import create_lattice_2nd_order_dynamic
gen = create_lattice_2nd_order_dynamic(size, omega, lattice_sep='-')
# type(gen) = HigherOrderPathGenerator
```

We can also convert a generator to a FON with `gen.to_FON()`, which discards all the higher–order transition probabilities.

We may also specify a `node_sort_key`, which defines the order of nodes and paths (which are first sorted by length and then by the individual nodes from start to end). This order is used when exporting the embedding as pandas DataFrame or decoding the probabilities as pandas Series.

Embedding We distinguish between *symmetric* and *asymmetric* embeddings. Asymmetric embeddings (e.g., NetMF and GraRep) calculate two embeddings `source_embedding` and `target_embedding`. Because they support both 'pairs of nodes' and 'mixed' interactions (see § 3.1.2), we assume that the source embeds paths (of length one in case of 'pairs of nodes'), and the target embeds nodes. Because pandas works better with an index of strings rather than tuples, we convert the paths (`path2str`) and nodes (`node2str`) - both are configurable with `repr` as default. As this gets confusing, we will use an EmbeddingView, which unifies both variants, and refers to both single nodes and paths as `key`.

Symmetric embeddings (e.g., DeepWalk and Node2vec) are restricted to 'pairs of nodes' interactions and calculate a single `embedding`, and there is no need to distinguish between key and node.

To calculate an embedding, instantiate an embedding class and call train().

```
from Embedding import HON_NetMF_Embedding
emb = HON_NetMF_Embedding(gen, dimension=128, pairwise=True)
%time emb.train(window_size=10, negative=1, optimized=True)
```

The Generic_SkipGram_Embedding imports the output of, e.g., LINE and Node2vec instead of calculating an embedding; see data/LINE/LINE.ipynb.

EmbeddingView and Visualization

```
from Visualizations import Visualization, EmbeddingView
ev = EmbeddingView(emb, use_source=True)
vis = ev.visualize_TSNE(random_state=1, n_iter=1000)
vis.plot1(hue='Class', figsize=(5,4), dpi=200)
#vis.annotate_node('xyz', 'XYZ', 'center', 'center')
vis.save_describe('tmp/plot.png', comment='...')
```

In the case of a HON Lattice 2D, we use Lattice2D_EmbeddingView instead of EmbeddingView, as the former provides additional functionality related to the grid structure.

The EmbeddingView is currently initialized after the embedding, but I better had the embedding provide these instances as properties `source` and `target` and tidied up the public interface. The HigherOrderPathGenerator would act as a factory and decide whether it represents a lattice.

C.2 HigherOrderPathGenerator.py

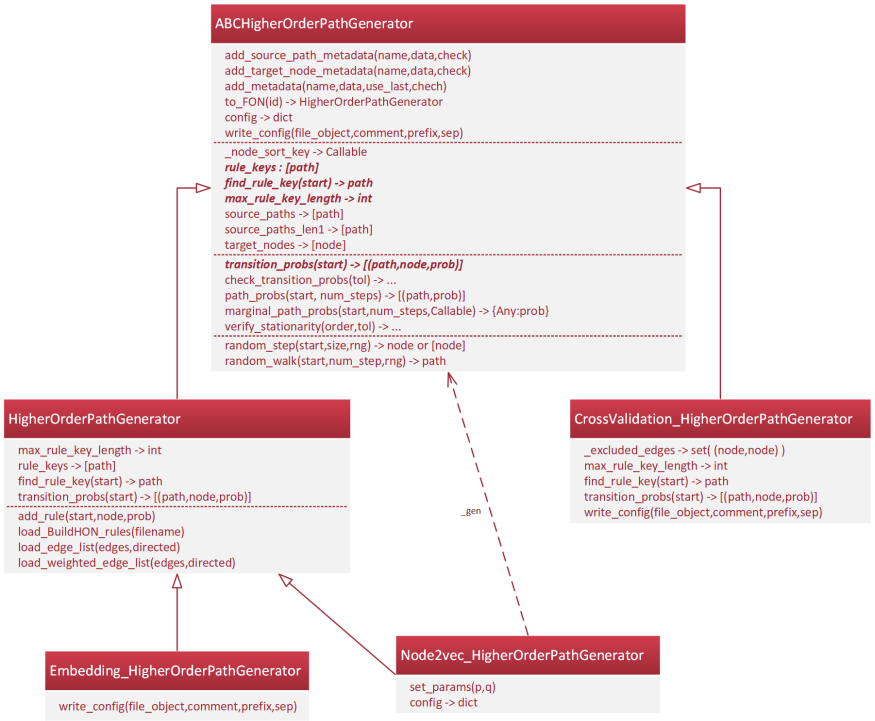


Figure C.1: UML class diagram for HigherOrderPathGenerator.py

These classes store the transition probabilities $T^{(k)}_{s_{1..k} \to t}$ and enumerate or simulate random walks.

```
from abc import ABC, abstractmethod
from typing import Any, Callable, Dict, List, Iterator, Tuple, Optional #
    https://mypy.readthedocs.io/en/stable/cheat_sheet_py3.html
from collections import defaultdict, Counter
import math
import numpy as np
import pandas as pd
#import random
import bisect

class ABCHigherOrderPathGenerator(ABC):
  "Generator for higher-order paths (abstract) - storage of rules is not
      implemented here"
  def __init__(self, node_sort_key=lambda node: node, id: Optional[str] =
      None, config: Dict[str,Any] = dict()):
    self._rules_cumulated = None
    self._node_sort_key = node_sort_key
    self._id = type(self).__name__ if id is None else id
    self._source_path_metadata = dict() # Dict[str, Dict[Tuple[Any,...],Any]]
    self._target_node_metadata = dict() # Dict[str, Dict[Any,Any]]
    self._config = config

  # metadata will be used in the visualizations
  def add_metadata(self, name, data: Dict[Any,Any], use_last: bool = True,
      check: bool = True):
    """
    Add metadata describing nodes for both target_nodes and source_paths.
    For the latter use either the first or last node of the path.
```

```
25      """
26      self.add_target_node_metadata(name, data, check)
27      index = -1 if use_last else 0
28      paths_data = { key: data.get(key[index], None) for key in self.
            source_paths }
29      self.add_source_path_metadata(name, paths_data, check=False)
30
31    def add_source_path_metadata(self, name: str, data: Dict[Tuple[Any,...],Any
            ], check: bool = True):
32      "Add␣metadata␣describing␣the␣source_paths"
33      if check:
34        assert len(set(data.keys()).difference(self.source_paths))==0, '
              metadata␣"%s"␣contains␣invalid␣keys' % name
35      self._source_path_metadata[name] = data
36
37    def add_target_node_metadata(self, name: str, data: Dict[Any,Any], check:
            bool = True):
38      "Add␣metadata␣describing␣the␣target_nodes"
39      if check:
40        assert len(set(data.keys()).difference(self.target_nodes))==0, '
              metadata␣"%s"␣contains␣invalid␣keys' % name
41      self._target_node_metadata[name] = data
42
43    @property
44    @abstractmethod
45    def rule_keys(self) -> Iterator[Tuple[Any,...]]:
46      pass
47
48    @abstractmethod
49    def find_rule_key(self, start: Tuple[Any,...]) -> Tuple[Any,...]:
50      pass
51
52    @property
53    @abstractmethod
54    def max_rule_key_length(self) -> int:
55      pass
56
57    @property
58    def source_paths(self) -> List[Tuple[Any,...]]:
59      sort_key = lambda path:(len(path), *map(self._node_sort_key, path))
60      "returns␣a␣sorted␣list␣of␣source␣paths␣of␣the␣transition␣robabilities␣(
            empty␣key␣is␣removed)"
61      return sorted({ key for key in self.rule_keys if len(key) >= 1 }, key=
            sort_key)
62
63    @property
64    def source_paths_len1(self) -> List[Tuple[Any,...]]:
65      "returns␣a␣sorted␣list␣of␣source␣paths␣of␣length␣1␣of␣the␣transition␣
            robabilities"
66      sort_key = lambda path:self._node_sort_key(path[0])
67      #return sorted({ (v,) for key in self.rule_keys for v in key }, key=
            sort_key) # safe & slower
68      return sorted({ key for key in self.rule_keys if len(key) == 1 }, key=
            sort_key)
69
70    @property
71    def target_nodes(self) -> List[Tuple[Any,...]]:
72      "returns␣a␣sorted␣list␣target␣nodes␣of␣the␣transition␣robabilities"
73      #return sorted({ next_node for key in self.rule_keys for _, next_node, _
            in self.transition_probs(key) }) # safe & slower
74      return sorted({ key[0] for key in self.rule_keys if len(key) == 1 }, key=
            self._node_sort_key) # this should also work, as otherwise random
            walks are broken.
75
76    @abstractmethod
77    def transition_probs(self, start: Tuple[Any,...]) -> Iterator[Tuple[Tuple[
            Any,...],Any,float]]:
78      pass
79
80    def check_transition_probs(self, tol=1e-14) -> Iterator[Tuple[Tuple[Any
            ,...],float]]:
81      "Verify␣that␣the␣transition␣probabilities␣sum␣up␣to␣one␣(with␣some␣
            numerical␣tolerance)."
82      for key in self.rule_keys:
```

```
83        total_prob = sum(prob for _,_,prob in self.transition_probs(key))
84        if abs(total_prob-1)>tol:
85          yield (key, total_prob)
86
87    def path_probs(self, start: Tuple[Any,...], num_steps: int = 1) -> Iterator
          [Tuple[Tuple[Any,...],float]]:
88      "Conditional probabilities of all paths starting with some given sequence
            of nodes."
89      if num_steps <= 0:
90        yield (start, 1.0)
91      else: #elif num_steps >= 1:
92        for outer_path, outer_prob in self.path_probs(start, num_steps-1):
93          for _, next_node, prob in self.transition_probs(outer_path):
94            yield ((*outer_path, next_node), outer_prob*prob)
95
96    def marginal_path_probs(self, start: Tuple[Any,...], num_steps: int = 1,
          projection: Callable[[Tuple[Any,...]] ,Any] = lambda x:x[-1]) -> Dict[
          Any,float]:
97      "Calculates the marginal probabilities of path_probs. May be used to
            calculate k-step transition probabilities."
98      probs = defaultdict(float)
99      for path,prob in self.path_probs(start,num_steps):
100       key = projection(path)
101       probs[key]+=prob
102     return probs
103
104   def verify_stationarity(self, order=2, tol=1e-14) -> Iterator[Tuple[Any,
          float,float,float]]:
105     """
106     Verify that the stationary distribution is indeed stationary with respect
            to the first order transition probabilities.
107
108     The (1st order) stationary distribution is stored as the transition
            probabilities of an empty start path.
109     """
110     mprobs = self.marginal_path_probs(start=(), num_steps=order+1, projection
            =lambda x:x[-1]) # node-distribution after a few steps
111     for _,node,prob in self.transition_probs(start=()): # stationary
            distribution
112       mprob = mprobs[node]
113       if abs(prob - mprob) > tol:
114         yield (node, prob, mprob, prob-mprob)
115
116   @property
117   def rules_cumulated(self):
118     "Cumulates the transition probabilities for random.choices"
119     if self._rules_cumulated is None:
120       rules2 = dict()
121       for start in self.rule_keys:
122         next_nodes = list()
123         cum_weights = list()
124         cum_prob = 0.0
125         for _, next_node, prob in self.transition_probs(start):
126           cum_prob += prob
127           next_nodes.append(next_node)
128           cum_weights.append(cum_prob)
129         rules2[start] = (next_nodes, cum_weights)
130       self._rules_cumulated = rules2
131     return self._rules_cumulated
132
133   def random_step(self, start: Tuple[Any,...], size=None, rng: Optional[np.
          random.Generator] = None) -> Any:
134     if rng is None:
135       rng = np.random.default_rng()
136     rule_key = self.find_rule_key(start)
137     if rule_key == () and start != (): # or assert start is a tuple
138       print('could not find a key matching %s' % start)
139     next_nodes,cum_weights = self.rules_cumulated[rule_key]
140     if size is None: # return scalar
141       x = rng.random(1)
142       # return random.choices(next_nodes, cum_weights=cum_weights, k=1)[0] #
            fine, but wrong RNG
143       return next_nodes[bisect.bisect_left(cum_weights, x[0], hi=len(
            cum_weights)-1)]
```

```
144       else: # return list
145         x = rng.random(size)
146         f = lambda y:next_nodes[bisect.bisect_left(cum_weights, y, hi=len(
                cum_weights)-1)]
147         return list(map(f, x))
148
149     def random_walk(self, start: Tuple[Any,...], num_steps:int = 1, rng:
            Optional[np.random.Generator] = None) -> Tuple[Any,...]:
150       if rng is None:
151         rng = np.random.default_rng()
152       walk = start
153       for _ in range(num_steps):
154         next_node = self.random_step(walk, size=None, rng=rng)
155         walk = (*walk,next_node)
156       return walk
157
158     def to_FON(self, id: Optional[str]  =None) -> "HigherOrderPathGenerator":
159       "Return␣a␣HigherOrderPathGenerator␣containing␣only␣first-order␣rules␣(and
              ␣the␣stationary␣distribution)"
160       if id is None:
161         id = self._id + '␣(FON)'
162       fon = HigherOrderPathGenerator(node_sort_key=self._node_sort_key, id=id,
              config=self._config)
163       if hasattr(self, 'creator'): # see Lattice2D_2nd_order_dynamic & required
              for Lattice2D_EmbeddingView
164         fon.creator = self.creator
165       for key in self.rule_keys:
166         if len(key)>1:
167           continue
168         for start, next_node, prob in self.transition_probs(key):
169           fon.add_rule(start, next_node, prob)
170       for name, data in self._source_path_metadata.items():
171         fon.add_source_path_metadata(name, { key: value for key,value in data.
              items() if len(key)==1 }, check=True)
172       fon._target_node_metadata = dict(self._target_node_metadata)
173       return fon
174
175     @property
176     def config(self):
177       "get␣configuration"
178       cfg=dict(init_class=self.__class__.__name__, init_id=self._id)
179       cfg.update(self._config)
180       return cfg
181
182     def write_config(self, file_object, comment: str = '', prefix: str = '',
            sep: str ='\t'):
183       if comment != '':
184         file_object.write(prefix + comment + '\n' + prefix + '\n')
185       file_object.write(prefix + 'HigherOrderPathGenerator:\n')
186       for k,v in self.config.items():
187         file_object.write(prefix + '%s%s%s\n' % (k,sep,v))
188
189 class HigherOrderPathGenerator(ABCHigherOrderPathGenerator):
190   "Generator␣for␣higher-order␣paths"
191   def __init__(self, node_sort_key=lambda node: node, id: Optional[str] =
            None, config: Dict[str,Any] = dict()):
192     super().__init__(node_sort_key, id, config)
193     self.rules = defaultdict(dict)
194     self._max_rule_key_length = 1 # find_rule_key relies on
            _max_rule_key_length > 0
195
196   @property
197   def max_rule_key_length(self) -> int:
198     return self._max_rule_key_length
199
200   def clear_rules(self):
201     "Clear␣the␣rules␣dictionary␣(incl.␣clean␣up)"
202     self._max_rule_key_length = 1
203     self.rules.clear()
204     self._rules_cumulated = None
205
206   def add_rule(self, start: Tuple[Any,...], next_node: Any, prob: float):
207     "Add␣entries␣to␣rules.␣(load_...␣methods␣must␣not␣manipulate␣the␣rules␣
            directly.)"
```

```
208     assert type(start)==tuple, 'argument␣"start"␣expected␣a␣tuple,␣but␣got␣%r
            ' % start
209     self._max_rule_key_length = max(len(start), self._max_rule_key_length)
210     self.rules[start][next_node] = prob
211
212   def load_BuildHON_rules(self, filename: str):
213     "Load␣output␣of␣BuildHON+"
214     with open(filename, 'r') as f:
215       for line in f:
216         items = line.split('␣')
217         assert (len(items)>=3) and (items[-3]=='=>'), 'invalid␣line␣"%s"' %
              line
218         key = tuple(map(int,items[:-3]))
219         self.add_rule(key, int(items[-2]), float(items[-1]))
220     print('%d␣rules␣read' % sum([len(v) for v in self.rules.values()]))
221
222   def load_edge_list(self, edges : Iterator[Tuple[Any,Any]], directed=True):
223     def add_weights(edges_: Iterator[Tuple[Any,Any]]) -> Iterator[Tuple[Any,
            Any,Any]]:
224       for u,v in edges_:
225         yield u,v,1 # add weight
226     self.load_weighted_edge_list(add_weights(edges), directed)
227
228   def load_weighted_edge_list(self, edges : Iterator[Tuple[Any,Any,Any]],
          directed=True):
229     if not directed:
230       def to_directed(edges_):
231         for u,v,w in edges_:
232           yield u,v,w
233           yield v,u,w
234       edges = to_directed(edges)
235     edges_list = list(edges) # iterating twice over edges does not work
236     out_weights = defaultdict(float)
237     for u,_,w in edges_list:
238       out_weights[u] += w
239     for u,v,w in edges_list:
240       self.add_rule((u,), v, w/out_weights[u])
241
242   @property
243   def rule_keys(self) -> Iterator[Tuple[Any,...]]:
244     return self.rules.keys()
245
246   def find_rule_key(self, start: Tuple[Any,...]) -> Tuple[Any,...]:
247     if (len(start) > self._max_rule_key_length): # and (self.
            _max_rule_key_length > 0):
248       start = start[-self._max_rule_key_length:]
249     while len(start)>0:
250       if start in self.rules:
251         return start
252       else:
253         start = start[1:]
254     return tuple()
255
256   def transition_probs(self, start: Tuple[Any,...]) -> Iterator[Tuple[Tuple[
          Any,...],Any,float]]:
257     rule_key = self.find_rule_key(start)
258     probs = self.rules[rule_key]
259     for next_node,prob in probs.items():
260       yield(rule_key, next_node, prob)
261
262 class Node2vec_HigherOrderPathGenerator(HigherOrderPathGenerator):
263   """
264   Adjusts the transition probabilities for the search bias of Node2vec.
265   Note, that Node2vec corresponds to DeepWalk with biased random walks.
266   """
267   def __init__(self, gen: ABCHigherOrderPathGenerator, p: float = 1, q: float
          = 1, id_format: Optional[str] = None):
268     # example: id_format = 'Node2vec(p={0}, q={1})'
269     super().__init__(gen._node_sort_key, id=gen._id, config=gen._config)
270     self._gen = gen
271     self._source_path_metadata = dict(gen._source_path_metadata)
272     self._target_node_metadata = dict(gen._target_node_metadata)
273     if hasattr(gen, 'creator'): # see Lattice2D_2nd_order_dynamic & required
            for Lattice2D_EmbeddingView
```

```
274       self.creator = gen.creator
275     self._id_format = id_format # if id_format is provided, _id will be
            overwritten
276     self.set_params(p, q)
277
278   def set_params(self, p: float = 1, q: float = 1):
279     "Sets␣the␣parameters␣for␣the␣search␣bias"
280     self._p = p
281     self._q = q
282     if self._id_format is not None:
283       self._id = self._id_format.format(p,q)
284     self.clear_rules()
285     rules_tmp = defaultdict(list)
286     # copy all rules
287     for key in self._gen.rule_keys:
288       rules_tmp[key] = list( (next_node, prob) for _, next_node, prob in self
              ._gen.transition_probs(key) )
289     # generate rules for all direct neighbors if they do not yet exist
290     direct_neighbors = set()
291     for key1 in self._gen.source_paths_len1: # key1 is a tuple containing one
            node
292       for _, key2, _ in self._gen.transition_probs(key1): # key2 is a node
293         key = (key1[0], key2)
294         direct_neighbors.add(key)
295         if key not in rules_tmp:
296           rules_tmp[key] = list( (next_node, prob) for _, next_node, prob in
                self._gen.transition_probs((key2,)) )
297     # add modified rules
298     for key, probs in rules_tmp.items():
299       if len(key) < 2:
300         for next_node, prob in probs:
301           self.add_rule(key, next_node, prob)
302       else:
303         prev_node = key[-2]
304         sum_probs = 0
305         new_probs = list()
306         for next_node, prob in probs:
307           # search bias
308           if next_node == prev_node:
309             prob /= p
310           elif (prev_node, next_node) not in direct_neighbors:
311             prob /= q
312           new_probs.append((next_node, prob))
313           sum_probs += prob
314         for next_node, prob in new_probs:
315           self.add_rule(key, next_node, prob / sum_probs)
316
317   @property
318   def config(self):
319     "get␣configuration"
320     cfg=dict(init_class=self.__class__.__name__, init_id=self._id, p=self._p,
            q=self._q)
321     cfg.update(self._config)
322     return cfg
323
324 class CrossValidation_HigherOrderPathGenerator(ABCHigherOrderPathGenerator):
325   """
326   Adjusts the transition probabilities for cross validation
327   """
328   def __init__(self, gen: ABCHigherOrderPathGenerator, excluded_edges :
          Iterator[Tuple[Any,Any]], id: Optional[str] = None):
329     self._gen = gen
330     self._excluded_edges = set(excluded_edges)
331     super().__init__(gen._node_sort_key, id, gen._config)
332     self._source_path_metadata = dict(gen._source_path_metadata)
333     self._target_node_metadata = dict(gen._target_node_metadata)
334     # init rules
335     self.rules = dict()
336     for start in gen.rule_keys:
337       if len(start)==0:
338         # todo: the stationary distribution must also be adjusted
339         self.rules[start]= { next_node: prob for _,next_node,prob in gen.
              transition_probs(start)}
340         continue
```

```
341       probs = list((next_node,prob) for _,next_node,prob in gen.
              transition_probs(start))
342       d = len(probs) # out degree
343       excluded = { next_node for next_node,_ in probs if (start[-1],next_node
              ) in self._excluded_edges }
344       excluded_prob = sum(prob for next_node,prob in probs if next_node in
              excluded)
345       self.rules[start] = { next_node: 1/d if next_node in excluded else prob
               * (1-len(excluded)/d) / (1-excluded_prob)
346           for next_node,prob in probs}
347
348   @property
349   def rule_keys(self) -> Iterator[Tuple[Any,...]]:
350     return self._gen.rule_keys
351
352   def find_rule_key(self, start: Tuple[Any,...]) -> Tuple[Any,...]:
353     return self._gen.find_rule_key(start)
354
355   @property
356   def max_rule_key_length(self) -> int:
357     return self._gen.max_rule_key_length
358
359   def transition_probs(self, start: Tuple[Any,...]) -> Iterator[Tuple[Tuple[
          Any,...],Any,float]]:
360     rule_key = self._gen.find_rule_key(start)
361     probs = self.rules[rule_key]
362     for next_node,prob in probs.items():
363       yield(rule_key, next_node, prob)
364
365   def write_config(self, file_object, comment: str = '', prefix: str = '',
          sep: str ='\t'):
366     self._gen.write_config(file_object=file_object, comment=comment, prefix=
          prefix, sep=sep)
367     file_object.write(prefix + '\n' + prefix + 'CrossValidation-
          HigherOrderPathGenerator:\n')
368     for k,v in self.config.items():
369       file_object.write(prefix + '%s%s%s\n' % (k,sep,v))
370
371 class Embedding_HigherOrderPathGenerator(HigherOrderPathGenerator):
372   "Generator␣based␣on␣the␣decode-method␣of␣an␣asymmetric␣embedding"
373   def __init__(self, emb: 'ABCAsymmetricEmbedding', use_neighborhood: bool =
          True, no_self_loops: bool = False, id: Optional[str] = None, **kwargs)
          :
374     config = dict(init_use_neighborhood=use_neighborhood, init_no_self_loops=
          no_self_loops, init_emb=emb._id, **kwargs)
375     super().__init__(emb._gen._node_sort_key, id=id, config=config)
376     gen = emb._gen
377     self._emb = emb
378     self._source_path_metadata = dict(gen._source_path_metadata)
379     self._target_node_metadata = dict(gen._target_node_metadata)
380     if hasattr(gen, 'creator'): # see Lattice2D_2nd_order_dynamic & required
          for Lattice2D_EmbeddingView
381       self.creator = gen.creator
382     str2nodes = { emb.node2str(node):node for node in emb.target_nodes }
383     for start in emb.source_paths:
384       for next_node_str,prob in emb.decode_path(start=start, use_neighborhood
              =use_neighborhood, no_self_loops=no_self_loops, normalize=True, **
              kwargs).items():
385         if prob>0:
386           self.add_rule(start, str2nodes[next_node_str], prob)
387     # todo: calculate stationary distribution from the first-order rules
388     for _,next_node,prob in gen.transition_probs(start=()): # copy stationary
           distribution from original generator
389       self.add_rule((), next_node, prob)
390
391   def write_config(self, file_object, comment: str = '', prefix: str = '',
          sep: str ='\t'):
392     self._emb.write_config(file_object=file_object, comment=comment, prefix=
          prefix, sep=sep)
393     file_object.write(prefix + '\n' + prefix + 'Embedding-
          HigherOrderPathGenerator:\n')
394     for k,v in self.config.items():
395       file_object.write(prefix + '%s%s%s\n' % (k,sep,v))
```

C.3 SyntheticNetworks.py

This creates a `HigherOrderPathGenerator` containing a HON Lattice 2D (§ 3.3).

```
from HigherOrderPathGenerator import HigherOrderPathGenerator

from collections import Counter
import numpy as np
import pandas as pd
import math

def create_lattice_2nd_order_dynamic(size: int=10, omega: float=0,
     lattice_sep: str='-', check: bool=False):
  creator = Lattice2D_2nd_order_dynamic(size, lattice_sep)
  return creator.create_generator(omega, check)

class Lattice2D_2nd_order_dynamic(object):
  def __init__(self, size: int=10, lattice_sep: str='-'):
    self.size=size
    self.lattice_sep = lattice_sep
    # '0-0' corresponds to the lower-left corner of the lattice. Format is (x,y).
    self.neighbor_funcs = { (-1,0): self.left, (1,0): self.right, (0,1): self.up, (0,-1): self.down }
    self.coord2nodes = { (x,y): '%i%s%i' % (x, lattice_sep, y) for x in range(size) for y in range(size) }
    self.node2coords = { n:c for c,n in self.coord2nodes.items() }

  def left(self, node:str)->str:
    x,y = node.split(self.lattice_sep)
    x = str(max(0, int(x)-1))
    return x + self.lattice_sep + y
  def right(self, node:str)->str:
    x,y = node.split(self.lattice_sep)
    x = str(min(self.size-1, int(x)+1))
    return x + self.lattice_sep + y
  def up(self, node:str)->str:
    x,y = node.split(self.lattice_sep)
    y = str(min(self.size-1, int(y)+1))
    return x + self.lattice_sep + y
  def down(self, node:str)->str:
    x,y = node.split(self.lattice_sep)
    y = str(max(0, int(y)-1))
    return x + self.lattice_sep + y

  @property
  def horizontal_edges1(self):
    return list( (u,v) for (u,v) in [(u,self.right(u)) for u in self.coord2nodes.values()] if u!=v )

  @property
  def vertical_edges1(self):
    return list( (u,v) for (u,v) in [(u,self.up(u)) for u in self.coord2nodes.values()] if u!=v )

  @property
  def horizontal_edges2(self):
    return list( ((u,v),(v,w)) for (u,v,w) in [(self.left(u),u,self.right(u)) for u in self.coord2nodes.values()] if u!=v and v!=w)

  @property
  def vertical_edges2(self):
    return list( ((u,v),(v,w)) for (u,v,w) in [(self.down(u),u,self.up(u)) for u in self.coord2nodes.values()] if u!=v and v!=w)

  def create_generator(self, omega: float=0, check: bool=False):
    assert abs(omega)<=1
    config = dict(code=__file__, init=self.__class__.__name__ + '.create_generator',
      size=self.size, lattice_sep=repr(self.lattice_sep), omega=omega)
    # create list of nodes and (undirected) edges
    # see also pathpy.generators.lattice_network(start=0, stop=size, dims=2)
    nodes = list(self.coord2nodes.values())
    undirected_edges = self.horizontal_edges1 + self.vertical_edges1
```

```
62      edges = undirected_edges + [(v,u) for u,v in undirected_edges]
63      degrees = Counter(u for u,v in edges)
64      sum_degrees = sum(degrees.values())
65
66      #node_sort_key = lambda x: tuple(map(int,x.split('-')))
67      node_sort_key = self.node2coords.__getitem__
68      latgen = HigherOrderPathGenerator(node_sort_key, 'Lattice2D(%d, omega=%f)
            ' % (self.size, omega), config)
69      latgen.creator = self
70      latgen.load_edge_list(edges, directed=True)
71      # add 2nd order rules for (u,v) -> w
72      for u in nodes:
73        latgen.add_rule((), u, degrees[u]/sum_degrees) # prob of stationary
              distribution
74        for (d1,f1) in self.neighbor_funcs.items():
75          v = f1(u)
76          if u == v: # cannot move / no self-loops
77            continue
78          for (d2,f2) in self.neighbor_funcs.items():
79            w = f2(v)
80            if v == w: # cannot move / no self-loops
81              continue
82            if v == f1(v): # if we cannot move twice in direction f1 from u, do
                  not change dynamic
83              d = 0.0
84            else:
85              d = d1[0]*d2[0] - d1[1]*d2[1] # horizontal speed-up and vertival
                    slow-down
86            latgen.add_rule((u, v), w, (1 + d*omega)/degrees[v])
87
88      latgen.add_source_path_metadata('key_len', { n: len(n) for n in latgen.
            source_paths }, check=False)
89      latgen.add_metadata('x_orig', { n: c[0] for c,n in self.coord2nodes.items
            () }, use_last=True, check=False)
90      latgen.add_metadata('y_orig', { n: c[1] for c,n in self.coord2nodes.items
            () }, use_last=True, check=False)
91      latgen.add_metadata('parity', { n: 'even' if sum(c)%2==0 else 'odd' for c
            ,n in self.coord2nodes.items() }, use_last=True, check=False)
92      direction = dict()
93      for key in latgen.source_paths:
94        if len(key) == 1:
95          direction[key]='none'
96        else:
97          c0 = self.node2coords[key[0]]
98          c1 = self.node2coords[key[-1]]
99          delta = (c1[0]-c0[0], c1[1]-c0[1])
100         direction[key]=self.neighbor_funcs[delta].__name__
101     latgen.add_source_path_metadata('direction', direction, check=False)
102     if check:
103       # check whether probabilities sum up to one
104       print(list(latgen.check_transition_probs(tol=1e-14)))
105       # verify that the 2nd order rules did not modify the stationary
              distribution
106       print(list(latgen.verify_stationarity(order=2, tol=1e-17)))
107     return latgen
```

C.4 Datasets.py

The SocioPattern networks (§ 4.2.2) are loaded with `init_generator`.

```
from HigherOrderPathGenerator import HigherOrderPathGenerator
import os

def init_generator(filename, tol1=1e-14, tol2=1e-15, verbose=False):
  filepath = 'data\\models\\'+filename
  with open(os.path.splitext(filepath)[0]+'.config', 'r') as f:
    model_config = { items[0]:' '.join(items[1:]) for items in [line.strip('\n').split('\t') for line in f.readlines() ] }
  if verbose:
    print('Configuration:')
    for k,v in model_config.items():
      print('%s: %s' % (k,v))
  model_config['filename'] = filename

  gen = HigherOrderPathGenerator(node_sort_key=int, id=os.path.splitext(filename)[0], config=model_config)
  gen.load_BuildHON_rules('data\\models\\'+filename)
  # add metadata
  metadata_filename = 'metadata_' + (filename.split('.')[0].split('_')[0]) + '.csv'
  metadata = dict()
  with open('data\\models\\' + metadata_filename, 'r') as f:
    for line in f:
      i,Ci = line.split()
      metadata[int(i)]=Ci
  if metadata_filename=='metadata_workplace.csv':
    gen.add_metadata('Department', metadata, use_last=True)
  elif metadata_filename=='metadata_primaryschool.csv':
    gen.add_metadata('Class', metadata, use_last=True)
    gen.add_metadata('Role', { i:'Teacher' if c=='Teachers' else 'Child' for i,c in metadata.items()})
  elif metadata_filename=='metadata_hospital.csv':
    gen.add_metadata('Status', metadata, use_last=True)
  elif metadata_filename=='metadata_temporal-clusters.csv' or metadata_filename=='metadata_shuffled-temporal-clusters.csv':
    gen.add_metadata('Color', metadata, use_last=True)
  else:
    raise Exception('unknown metadata_filename')
  print(list(gen.check_transition_probs(tol1)))
  print(list(gen.verify_stationarity(1, tol2)))
  #print(list(gen.verify_stationarity(2, tol2))) # this does not hold
  return gen
```

C.5 Embedding.py

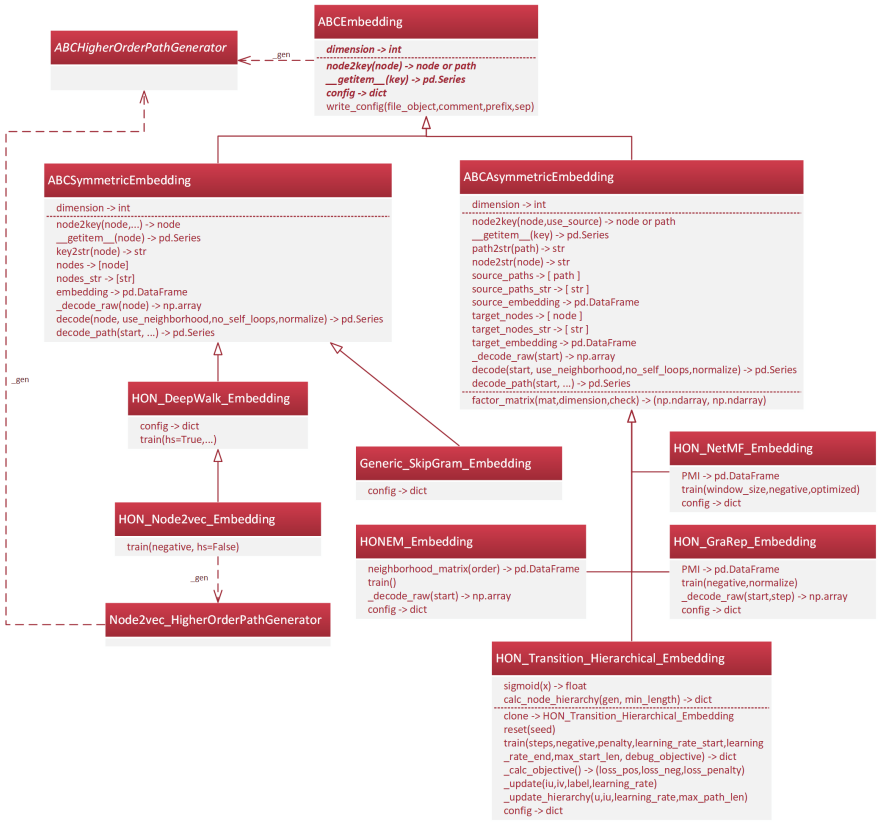


Figure C.2: UML class diagram for Embedding.py

All the embeddings mentioned in § 3.2 are implemented here.

```
1  from abc import ABC, abstractmethod
2  from typing import Any, Callable, Dict, Iterator, List, Tuple, Optional,
       Union
3  import warnings
4  import numpy as np
5  import pandas as pd
6  import math
7  import collections
8  import os
9
10 from HigherOrderPathGenerator import ABCHigherOrderPathGenerator,
       HigherOrderPathGenerator, Node2vec_HigherOrderPathGenerator
11 from gensim.models import Word2Vec
12 from scipy.linalg import svd
13 from sklearn.decomposition import TruncatedSVD
14 from zlib import adler32, crc32
15 import sys
16
17 # While the public interface of the embedding classes presents the
18 # embedding matrices as pandas DataFrames, they are internally stored
19 # in numpy ndarrays together with dictionaries for the row and column
20 # names. Besides better performance, the representation of paths as
21 # tuples did not fit well with pandas - where the paths had to be converted
22 # into strings. (And converting paths into str or int is basically the
23 # same in terms of readibility of the code.)
```

```
24
25 class ABCEmbedding(ABC):
26   def __init__(self, gen: ABCHigherOrderPathGenerator, dimension: int,
         symmetric:bool=True):
27     self._gen = gen
28     self._symmetric = symmetric
29     self._dimension = dimension
30     self._id = type(self).__name__
31
32   @property
33   @abstractmethod
34   def dimension(self):
35     "effective␣dimension␣of␣embedding"
36     pass
37
38   @abstractmethod
39   def node2key(self, node: Any, use_source: bool=False):
40     pass
41
42   @abstractmethod
43   def __getitem__(self, key) -> pd.Series:
44     "Returns␣a␣row␣of␣the␣embedding␣matrix."
45     pass
46
47   @abstractmethod
48   def decode_path(self, start: Tuple[Any,...], **kwargs) -> pd.Series:
49     pass
50
51   @property
52   @abstractmethod
53   def config(self):
54     "get␣configuration"
55     pass
56
57   def write_config(self, file_object, comment: str = '', prefix: str = '',
         sep: str ='\t'):
58     self._gen.write_config(file_object=file_object, comment=comment, prefix=
          prefix, sep=sep)
59     file_object.write(prefix + '\n' + prefix + 'Embedding:\n')
60     for k,v in self.config.items():
61       file_object.write(prefix + '%s%s%s\n' % (k,sep,v))
62
63 class ABCSymmetricEmbedding(ABCEmbedding):
64   def __init__(self, gen: ABCHigherOrderPathGenerator, dimension: int, nodes:
        Iterator[Any]):
65     super().__init__(gen, dimension, symmetric=True)
66     self._nodes = { n:i for i,n in enumerate(nodes) }
67     self._embedding = np.zeros(shape=(len(self._nodes), dimension))
68     self.key2str = repr
69
70   def __getitem__(self, key) -> pd.Series:
71     idx = self._nodes[key]
72     return pd.Series(self._embedding[idx,:], name=self.key2str(key))
73
74   def node2key(self, node: Any, use_source: bool=False):
75     return node
76
77   @property
78   def nodes(self) -> Iterator[Any]:
79     return self._nodes.keys()
80
81   @property
82   def nodes_str(self) -> List[str]:
83     return list(map(self.key2str, self.nodes))
84
85   @property
86   def embedding(self) -> pd.DataFrame:
87     return pd.DataFrame(self._embedding, index=self.nodes_str)
88
89   @property
90   def dimension(self):
91     "effective␣dimension␣of␣embedding"
92     return self._embedding.shape[1]
93
```

```
94    def _decode_raw(self, node: Any, step = None) -> np.array:
95      """Calculate the transition probabilities for start from the embedding.
96      Default implementation assumes the Skip-gram model.
97      (The parameter step is not used.)
98      """
99      return np.exp(self._embedding[self._nodes[node]] @ self._embedding.T)
100
101   def decode(self, node: Any, use_neighborhood: bool = False, no_self_loops:
          bool = False, normalize: bool = True, **kwargs) -> pd.Series:
102     """Calculate probabilities from the embedding.
103     The parameter node must contain a value from self.nodes.
104
105     Parameters
106     ----------
107
108     node : Any
109
110       key of the embedding.
111
112     use_neighborhood : bool, optional (default = False)
113
114       restrict the probabilities to the neighborhood (using knowledge about
              network toology).
115
116     no_self_loops : bool, optional (default = False)
117
118       set the probability of a self loop to zero.
119
120     normalize : bool, optional (default = True)
121
122       normalize the probabilities, such that they sum up to one.
123
124     **kwargs are passed to _decode_raw(node)
125     """
126     data = self._decode_raw(node=node, **kwargs)
127     if use_neighborhood:
128       data_n = np.zeros(len(data))
129       for _,next_node,_ in self._gen.transition_probs(start=(node,)):
130         i = self._nodes[next_node]
131         data_n[i] = data[i]
132       data = data_n
133     if no_self_loops:
134       data[self._nodes[node]] = 0
135     if normalize:
136       data = data / max(1e-16, data.sum())
137     return pd.Series(data, index=self.nodes_str, name=self.key2str(node))
138
139   def decode_path(self, start: Tuple[Any,...], **kwargs) -> pd.Series:
140     "Returns␣decode()␣for␣the␣last␣node␣of␣the␣path."
141     return self.decode(start[-1], **kwargs)
142
143 class ABCAsymmetricEmbedding(ABCEmbedding):
144   def __init__(self, gen: ABCHigherOrderPathGenerator, dimension: int,
          source_paths: Iterator[Tuple[Any,...]], target_nodes: Iterator[Any]):
145     super().__init__(gen, dimension, symmetric=False)
146     self._source_paths = { n:i for i,n in enumerate(source_paths) }
147     self._target_nodes = { n:i for i,n in enumerate(target_nodes) }
148     self._source_embedding = np.zeros(shape=(len(self._source_paths),
            dimension))
149     self._target_embedding = np.zeros(shape=(len(self._target_nodes),
            dimension))
150     self.path2str = repr
151     self.node2str = repr
152
153   def __getitem__(self, key) -> pd.Series:
154     """
155     If the key contains a node, a row of the target_embedding is returned.
156     And if the key contains a tuple of nodes, a row of the source_embedding
            matrix is returned instead.
157     """
158     idx = self._target_nodes.get(key, None)
159     if idx is not None:
160       return pd.Series(self._target_embedding[idx,:], name=self.node2str(key)
              )
```

```
161       idx = self._source_paths.get(key, None)
162       if idx is not None:
163         return pd.Series(self._source_embedding[idx,:], name=self.path2str(key)
                )
164       else:
165         raise IndexError('Unknown key %s' % key, key=key)
166
167     def node2key(self, node: Any, use_source: bool=False):
168       return (node,) if use_source else node
169
170     @property
171     def source_paths(self) -> Iterator[Tuple[Any,...]]:
172       return self._source_paths.keys()
173
174     @property
175     def source_paths_str(self) -> List[str]:
176       return list(map(self.path2str, self.source_paths))
177
178     @property
179     def target_nodes(self) -> Iterator[Any]:
180       return self._target_nodes.keys()
181
182     @property
183     def target_nodes_str(self) -> List[str]:
184       return list(map(self.node2str, self.target_nodes))
185
186     @property
187     def source_embedding(self) -> pd.DataFrame:
188       return pd.DataFrame(self._source_embedding, index=self.source_paths_str)
189
190     @property
191     def target_embedding(self) -> pd.DataFrame:
192       return pd.DataFrame(self._target_embedding, index=self.target_nodes_str)
193
194     @property
195     def dimension(self):
196       "effective dimension of embedding"
197       return self._source_embedding.shape[1]
198
199     def _decode_raw(self, start: Tuple[Any,...], step = None) -> np.array:
200       """Calculate the transition probabilities for start from the embedding.
201       Default implementation assumes the Skip-gram model.
202       (The parameter step is not used.)
203       """
204       return np.exp(self._source_embedding[self._source_paths[start]] @ self.
              _target_embedding.T)
205
206     def decode(self, start: Tuple[Any,...], use_neighborhood: bool = False,
           no_self_loops: bool = False, normalize: bool = True, **kwargs) -> pd.
           Series:
207       """Calculate probabilities from the embedding.
208       The parameter start must contain a value from self.source_paths.
209
210       Parameters
211       ----------
212
213       start : Tuple[Any, ...]
214
215         key of the source embedding.
216
217       use_neighborhood : bool, optional (default = False)
218
219         restrict the probabilities to the neighborhood (using knowledge about
              the first-order network topology).
220
221       no_self_loops : bool, optional (default = False)
222
223         set the probability of a self loop to zero.
224
225       normalize : bool, optional (default = True)
226
227         normalize the probabilities, such that they sum up to one.
228
229       **kwargs are passed to _decode_raw(start)
```

```
230     """
231     data = self._decode_raw(start=start, **kwargs)
232     if use_neighborhood:
233       data_n = np.zeros(len(data))
234       for _,next_node,_ in self._gen.transition_probs(start=start[-1:]):
235         i = self._target_nodes[next_node]
236         data_n[i] = data[i]
237       data = data_n
238     if no_self_loops:
239       data[self._target_nodes[start[-1]]] = 0
240     if normalize:
241       data = data / max(1e-16, data.sum())
242     return pd.Series(data, index=self.target_nodes_str, name=self.path2str(
          start))
243
244   def decode_path(self, start: Tuple[Any,...], **kwargs) -> pd.Series:
245     "Finds␣the␣matching␣source_path␣and␣returns␣decode()"
246     if start in self._source_paths:
247       return self.decode(start, **kwargs)
248     if (len(start) > self._gen.max_rule_key_length): # and (self._gen.
          max_rule_key_length > 0):
249       start = start[-self._gen.max_rule_key_length:]
250     while len(start)>0:
251       if start in self._source_paths:
252         return self.decode(start, **kwargs)
253       else:
254         start = start[1:]
255     return None
256
257   @staticmethod
258   def factor_matrix(mat: np.ndarray, dimension: int=None, check=False)->
          Tuple[np.ndarray, np.ndarray]:
259     "Factor␣matrix␣using␣(truncated)␣SVD"
260     if (dimension is None) or (dimension >= min(mat.shape)): # SVD
261       dimension = min(mat.shape)
262       U,S,Vh = svd(mat)
263       sqrtS = S**0.5
264       if mat.shape[0] > mat.shape[1]:
265         U = U[:, :dimension:] # U[:,:1] drops 2nd dimension (squeezing),
              while U[:, :1:] does not
266       elif mat.shape[0] < mat.shape[1]:
267         Vh = Vh[:dimension:, :]
268       source_embedding = U * (sqrtS[np.newaxis,:])
269       target_embedding = np.transpose((sqrtS[:,np.newaxis]) * Vh)
270     else:
271       tsvd = TruncatedSVD(n_components=dimension, random_state=1)
272       tsvd.fit(mat) # returns svd
273       W = tsvd.fit_transform(mat) # = U * Sigma, shape=(mat.shape[0],
            dimension)
274       H = tsvd.components_ # = V', shape=(dimension, mat.shape[1])
275       # checking factorization
276       #print('diff %g' % abs(mat - W@H).max())
277       S = np.linalg.norm(W,axis=0) # len(S) == dimension
278       S = S.clip(min=1e-20) # avoid division by zero
279       source_embedding = W / ((S**0.5)[np.newaxis,:]) # = W @ np.diag(S
            **-0.5)
280       target_embedding = np.transpose(H * ((S**0.5)[:,np.newaxis])) # = (np.
            diag(S**0.5) @ H).T
281     if check: # checking factorization
282       error = np.linalg.norm(mat - source_embedding @ target_embedding.
            transpose())
283       print('Approximation␣error␣%g' % (error/np.linalg.norm(mat)))
284     return (source_embedding, target_embedding)
285
286 class Generic_SkipGram_Embedding(ABCSymmetricEmbedding):
287   "Wrapper␣for␣embeddings␣based␣on␣the␣skip-gram␣model␣calculated␣externally␣
        (e.g.,␣LINE,␣Node2vec)"
288   def __init__(self, gen: ABCHigherOrderPathGenerator, emb_path : str, id:
        str = None, parse_node=lambda x:x, binary: bool=False, config=dict()):
289     assert binary==False, 'not␣implemented'
290     with open(emb_path, 'r') as f:
291       dims = f.readline().split('␣')
292       emb = pd.read_csv(f, sep='␣', header=None, comment='%', index_col=0)
293     if all(emb[emb.columns[-1]].isna()):
```

```
294       # The output of LINE has a training space resulting in a columns with
              NaN; Node2vec does not.
295       emb.drop(emb.columns[-1], axis=1, inplace=True)
296     super().__init__(gen, dimension=int(dims[1]), nodes=gen.target_nodes)
297     self._id = os.path.splitext(os.path.split(emb_path)[-1])[0] if id is None
            else id
298     self._emb_path = emb_path
299     #self._binary = binary
300     assert self._embedding.shape == emb.values.shape, f'shapes␣do␣not␣match:␣
            {self._embedding.shape}!={emb.values.shape}'
301     # verify row names and sort rows
302     emb.index = emb.index.map(parse_node)
303     keys_emb = set(emb.index)
304     keys_gen = set(gen.target_nodes)
305     if len(keys_emb.symmetric_difference(keys_gen))>0:
306       print('keys␣do␣not␣match␣(did␣you␣specify␣parse_node?)')
307       print('only␣in␣embedding:', keys_emb-keys_gen)
308       print('only␣in␣generator:', keys_gen-keys_emb)
309       assert False
310     ##emb.sort_index(axis=0, inplace=True, key=gen._node_sort_key) # requires
            pandas 1.1 and _node_sort_key must be vectorized
311     emb_sort = pd.Series(data=emb.index, index=emb.index).map(gen.
            _node_sort_key).sort_values().index
312     emb = emb.loc[emb_sort]
313     self._embedding = emb.values
314     self._config = config
315
316   def train(self):
317     pass
318
319   @property
320   def config(self):
321     "get␣configuration"
322     cfg = dict(init_class=self.__class__.__name__, init_gen=self._gen._id,
            init_emb_path = self._emb_path, init_id=self._id)#, init_binary=self
            ._binary)
323     cfg.update(self._config)
324     return cfg
325
326 class HON_DeepWalk_Embedding(ABCSymmetricEmbedding):
327   """
328   Adapts [DeepWalk] to random walks in higher order models, see
329   [DeepWalk] Perozzi B., Al-Rfou R., and Skiena S. (2014)
330   'Deepwalk: Online learning of social representations',
331   https://doi.org/10.1145/2623330.2623732
332
333   Uses gensim.models.Word2Vec for the embedding, which treats the random
334   walks as bidirectional. Hence, the embedding is symmetric.
335   """
336   def __init__(self, gen: ABCHigherOrderPathGenerator, dimension: int,
            reuse_walks: bool = False):
337     super().__init__(gen, dimension, gen.target_nodes)
338     self._id = 'DeepWalk'
339     self._reuse_walks = reuse_walks
340     self._walks = None
341     self._training_config= {'num_walks': 0, 'walk_length': 0, 'random_seed':
            0}
342
343   # https://stackoverflow.com/questions/34831551
344   # Word2vec uses the hash function to initialize the vector for each word,
          no need for a cryptographic hash function
345   if sys.hash_info.width == 64:
346     @staticmethod
347     def str_hash(data:str) -> int:
348       "deterministic␣hash␣function␣for␣Word2Vec␣(64␣bit)"
349       b = data.encode()
350       return adler32(b) + (crc32(b) << 32)
351   else:
352     @staticmethod
353     def str_hash(data:str) -> int:
354       "deterministic␣hash␣function␣for␣Word2Vec␣(32␣bit)"
355       return crc32(data.encode())
356
357   def train(self, num_walks:int=100, walk_length:int=80, window_size=10,
```

```
        num_iter:int=1, min_count:int=0,
358       hs:bool = True, negative:int=5, workers:int=1, random_seed=None,
              replace_hash:bool=True, **kwargs):
359     # reuse_walks allows for efficiently evaluating different params
360     # of Word2vec (e.g., negative, num_iter).
361     # walks depend only on num_walks, walk_length, and random_seed.
362     tc = self._training_config # config of previous training
363     can_reuse_walks = tc['num_walks']==num_walks and tc['walk_length']==
            walk_length and tc['random_seed']==random_seed
364     self._training_config = dict(num_walks=num_walks, walk_length=walk_length
            , window_size=window_size,
365         num_iter=num_iter, min_count=min_count, hs=hs, negative=negative,
                workers=workers,
366         random_seed=random_seed, replace_hash=replace_hash, **kwargs)
367     if (not can_reuse_walks) or (self._walks is None):
368       walks = list()
369       rng = np.random.default_rng(random_seed) # for shuffle and random walks
370       for _ in range(num_walks):
371         nodes_s = list(self.nodes)
372         rng.shuffle(nodes_s)
373         for node in nodes_s:
374           walks.append(self._gen.random_walk(start=(node,), num_steps=
                walk_length, rng=rng))
375       walks = [list(map(str, walk)) for walk in walks]
376       if self._reuse_walks:
377         self._walks = walks
378     else:
379       walks = self._walks
380     if replace_hash:
381       if 'hashfxn' in kwargs:
382         print('The␣parameter␣hashfxn␣is␣ignored␣because␣of␣replace_hash=True'
              )
383       kwargs['hashfxn'] = self.str_hash
384     model = Word2Vec(walks, size=self._dimension, window=window_size, sg=1,
            min_count=min_count, iter=num_iter,
385       hs=1 if hs else 0, negative=0 if hs else negative, workers=workers,
            seed=random_seed, **kwargs)
386     self.model = model # debug
387     for v,iv in self._nodes.items():
388       self._embedding[iv,:] = model.wv[str(v)]
389
390   @property
391   def config(self):
392     "get␣configuration"
393     cfg = dict(init_class=self.__class__.__name__, init_gen=self._gen._id,
            init_dimension=self._dimension, init_id=self._id)
394     cfg.update(self._training_config)
395     return cfg
396
397 class HON_Node2vec_Embedding(HON_DeepWalk_Embedding):
398   """
399   Adapts [Node2vec] to random walks in higher order models, see
400   [Node2vec] Grover A. and Leskovec J. (2016)
401   'node2vec: Scalable Feature Learning for Networks', https://doi.org
          /10.1145/2939672.2939754
402
403   The differences between Node2vec and DeepWalk are:
404   - Node2vec uses biased random walks, implemented in
          Node2vec_HigherOrderPathGenerator
405   - Both rely on gensim.models.Word2Vec, but Node2vec uses negative sampling
          and DeepWalk uses hierarchical softmax.
406   """
407   def __init__(self, gen: ABCHigherOrderPathGenerator, dimension: int = 128,
          p: float = 1, q: float = 1, reuse_walks:bool=False):
408     gen_n2v = Node2vec_HigherOrderPathGenerator(gen, p, q)
409     super().__init__(gen_n2v, dimension, reuse_walks)
410     self._id = 'Node2vec(p={0},␣q={1})'.format(p,q)
411
412   def set_params(self, p: float = 1, q: float = 1):
413     self.gen.set_params(p, q) # do not call this directly
414     self._id = 'Node2vec(p={0},␣q={1})'.format(p,q)
415     self._walks = None
416     #self._embedding = np.zeros(self._embedding.shape) # reset embedding
417
```

```
418   def train(self, negative:int=5, hs:bool = False, **kwargs): # Node2Vec uses
          negative sampling by default
419     super().train(hs=hs, negative=negative, **kwargs)
420
421 class HONEM_Embedding(ABCAsymmetricEmbedding):
422   """
423   Calculates the embedding according to
424   [HONEM] Saebi M., Ciampaglia G., Kaplan L., and Chawla N. (2019)
425   'HONEM: Network Embedding Using Higher-Order Patterns in Sequential Data',
          arXiv:1908.05387
426
427   The instance of class HigherOrderPathGenerator is assumed to contain the
          rules detected by BuildHON+, see
428   [BuildHON+] Xu J., Saebi M., Ribeiro B., Kaplan L., and Chawla N. (2017)
429   'Detecting Anomalies in Sequential Data with Higher-order Networks', arXiv
          :1712.09658
430   """
431   def neighborhood_matrix(self, order: int=1, sort=True) -> pd.DataFrame:
432     "Calculates␣the␣'v-th␣order␣neighborhood␣matrix'␣defined␣in␣the␣[HONEM]␣
          paper"
433     keys = [key for key in self._gen.source_paths if len(key)==order]
434     data = [(self.path2str((key[0],)), self.node2str(next_node), prob)
435         for key in keys for _, next_node, prob in self._gen.transition_probs(
              key)]
436     data_df = pd.DataFrame(data, columns=['src','trg','prob'])
437     res = data_df.groupby(['src','trg']).mean().unstack(fill_value=0)
438     res.columns = [c[1] for c in res.columns] # c[0] == 'prob'
439
440     # ensure that neighborhood matrices of different orders have identical
          indices and columns
441     all_source_nodes = { self.path2str((key[0],)) for key in self._gen.
          source_paths_len1 }
442     for v in all_source_nodes.difference(res.index):
443       res.loc[v]=0
444     all_target_nodes = { self.node2str(node) for node in self._gen.
          target_nodes }
445     for v in all_target_nodes.difference(res.columns):
446       res[v]=0
447
448     if sort:
449       res = res.loc[self.source_paths_str]
450       res = res[self.target_nodes_str]
451     return res
452
453   def __init__(self, gen: HigherOrderPathGenerator, dimension: int):
454     super().__init__(gen, dimension, gen.source_paths_len1, gen.target_nodes)
455     self._id = 'HONEM'
456
457   def train(self):
458     # calculate neighborhood matrix
459     neighborhood = self.neighborhood_matrix(1)
460     if self._gen.max_rule_key_length > 1:
461       for order in range(2, self._gen.max_rule_key_length+1):
462         neighborhood = neighborhood + math.exp(1-order) * self.
              neighborhood_matrix(order, sort=False)
463     # sort
464     neighborhood = neighborhood.loc[self.source_paths_str]
465     neighborhood = neighborhood[self.target_nodes_str]
466
467     source_embedding, target_embedding = self.factor_matrix(neighborhood.
          values, self._dimension)
468     self._source_embedding = source_embedding
469     self._target_embedding = target_embedding
470
471   def _decode_raw(self, start: Tuple[Any,...], step = None) -> np.array:
472     "Calculate␣the␣transition␣probabilities␣for␣start␣from␣the␣embedding.␣(
          HONEM␣does␣not␣use␣the␣skip-gram␣model)"
473     assert type(start) is tuple and len(start)==1, 'start␣must␣be␣a␣tuple␣of␣
          length␣1'
474     return self._source_embedding[self._source_paths[start]] @ self.
          _target_embedding.T
475
476   @property
477   def config(self):
```

```
478     "get␣configuration"
479     return dict(init_class=self.__class__.__name__, init_gen=self._gen._id,
            init_dimension=self._dimension, init_id=self._id)
480
481 class HON_NetMF_Embedding(ABCAsymmetricEmbedding):
482   """
483   Adapts [NetMF] to random walks in a higher-order model. (Additionally, the
          assumption of undirected graphs is dropped.)
484
485   [NetMF] Qiu, Jiezhong and Dong, Yuxiao and Ma, Hao and Li, Jian and Wang,
          Kuansan and Tang, Jie (2018)
486   'Network Embedding as Matrix Factorization: Unifying DeepWalk, LINE, PTE,
          and Node2vec', https://doi.org/10.1145/3159652.3159706
487   """
488   def __init__(self, gen: ABCHigherOrderPathGenerator, dimension: int,
          pairwise=False):
489     source_paths = gen.source_paths_len1 if pairwise else gen.source_paths
490     super().__init__(gen, dimension, source_paths, gen.target_nodes)
491     self._id = 'NetMF_pairs' if pairwise else 'NetMF_paths'
492     self._pairwise = pairwise
493     self._training_config= {}
494     # caching the PMI calculation
495     self._window_size = 0
496     self._PMI = None
497
498   def train(self, window_size:int, negative:int=1, optimized:bool=True):
499     assert window_size > 0, 'window_size␣must␣be␣a␣positive␣integer'
500     self._training_config = dict(window_size=window_size, negative=negative,
            optimized=optimized)
501     if window_size != self._window_size:
502       # calculate pointwise mutual information (PMI) - this takes some time
503       PMI = np.zeros(shape=(len(self.source_paths), len(self.target_nodes)))
504       idx = { v:i for i,v in enumerate(self.target_nodes) }
505       # stationary distribution (for target nodes)
506       sd = { v:p for _,v,p in self._gen.transition_probs(start=()) }
507       if optimized:
508         # We want to enumerate all paths and aggregate the visiting
509         # probabilities of the indiviidual nodes. However, enumerating the
510         # paths in a depth-first style is generally expensive (exponential
511         # growth in the length of number of paths). Instead use a breadth-
512         # first enumeration of the paths. Because of the higher-order Markov
513         # property of the transition probabilites, we need only to keep track
514         # of the last k=max_rule_key_length nodes visited. This avoids the
515         # exponential growth.
516         # There are max. (#nodes ** max_rule_key_length) different subpaths.
517         max_len = self._gen.max_rule_key_length
518         for iu,u in enumerate(self.source_paths):
519           paths = {u: 1/window_size}
520           for _ in range(window_size):
521             paths_new = collections.defaultdict(float)
522             for source,source_prob in paths.items():
523               for _,next_node,prob in self._gen.transition_probs(source):
524                 prob_new = source_prob * prob
525                 PMI[iu,idx[next_node]] += prob_new / sd[next_node]
526                 source_new = self._gen.find_rule_key((*source, next_node))
527                 paths_new[source_new] += prob_new
528             paths = paths_new
529       else:
530         # slower implementation enumerating all paths (debugging only)
531         for iu,u in enumerate(self.source_paths):
532           for path,prob in self._gen.path_probs(u, num_steps = window_size):
533             prob_div = prob / window_size # = probability of this path
                  conditional to start=u, divided by window_size
534             for v in path[len(u):]: # skip the start from the path
535               iv = idx[v]
536               PMI[iu,iv] += prob_div / sd[v]
537       # cache calculation
538       self._window_size = window_size
539       self._PMI = PMI
540     self._negative = negative
541     M = self._PMI / negative
542     mat = np.log(M.clip(1))
543     source_embedding, target_embedding = self.factor_matrix(mat, self.
          _dimension)
```

```
544       self._source_embedding = source_embedding
545       self._target_embedding = target_embedding
546
547     @property
548     def PMI(self) -> pd.DataFrame:
549       return pd.DataFrame(self._PMI, index=self.source_paths_str, columns=self.
              target_nodes_str)
550
551     @property
552     def config(self):
553       "get␣configuration"
554       cfg = dict(init_class=self.__class__.__name__, init_gen=self._gen._id,
              init_dimension=self._dimension, init_id=self._id, init_pairwise=self
              ._pairwise)
555       cfg.update(self._training_config)
556       return cfg
557
558   class HON_GraRep_Embedding(ABCAsymmetricEmbedding):
559     """
560     Adapts [GraRep] to random walks in a higher-order model.
561
562     [GraRep] Cao, Shaosheng and Lu, Wei and Xu, Qiongkai (2015)
563     'GraRep: Learning Graph Representations with Global Structural Information
            ', https://doi.org/10.1145/2806416.2806512
564     """
565     def __init__(self, gen: ABCHigherOrderPathGenerator, dimension: int,
            num_steps: int = 4, pairwise:bool=True, neg_stationary:bool=False):
566       """
567       Init
568
569       Parameters
570       ----------
571       dimension : int, >0
572         The dimension of the embedding for each individual step.
573         Effectively, the dimension is num_steps * dimension.
574
575       num_steps : int, >0 (default = 4)
576         ..
577
578       pairwise : bool (default = True)
579         Indicates whether nodes (True) or paths are embedded.
580
581       neg_stationary : bool (default = False)
582         The PMI is divided by the corresponding averages (False, as in GraRep)
583         or by the stationary distribution (True, as in HON NetMF).
584       """
585       source_paths = gen.source_paths_len1 if pairwise else gen.source_paths
586       self._dimension_per_step = dimension = min(dimension, len(gen.
              target_nodes)) # dimension returned by factor_matrix()
587       super().__init__(gen, self._dimension_per_step * num_steps, source_paths,
              gen.target_nodes)
588       self._dimension = dimension # specified dimension
589       self._id = 'GraRep_pairs' if pairwise else 'GraRep_paths'
590       self._pairwise = pairwise
591       self._num_steps = num_steps
592       self._neg_stationary = neg_stationary # distribution for negative samples
              : stationary (True) or averages (False)
593       self._training_config= {}
594       # caching the PMI calculation
595       self._PMI = None
596
597     def train(self, negative: int = 1, normalize: bool = False):
598       # optimized PMI calculation from HON_NetMF_Embedding (with small
              modifications).
599       # HON_NetMF_Embedding._PMI matches self._PMI.mean(axis=2) if window_size
              equals num_steps and neg_stationary=True
600       self._training_config = dict(negative=negative, normalize=normalize)
601       if self._PMI is None:
602         idx = { v:i for i,v in enumerate(self.target_nodes) }
603         PMI = np.zeros(shape=(len(self.source_paths), len(self.target_nodes),
              self._num_steps))
604         max_len = self._gen.max_rule_key_length
605         for iu,u in enumerate(self.source_paths):
606           paths = {u: 1} # 1 instead of 1/sindow_size in NetMF
```

```
607         for step0 in range(self._num_steps): # step0 = step - 1
608           paths_new = collections.defaultdict(float)
609           for source,source_prob in paths.items():
610             for _,next_node,prob in self._gen.transition_probs(source):
611               prob_new = source_prob * prob
612               PMI[iu, idx[next_node], step0] += prob_new
613               source_new = self._gen.find_rule_key((*source, next_node))
614               paths_new[source_new] += prob_new
615           paths = paths_new
616       # The PMI values calculated above correspond to p(c|w), and we stil
617       # have to divide it by p(c) - or similar. GraRep divides it by
618       # sum_w p(c|w) / |V|, see Table 1 in [GraRep]. (The division by the
619       # number of nodes |V| is borrowed from the parameter beta.)
620       # neg_stationary=True instead divides by the stationary distribution
621       # for comparison with NetMF.
622       averages = np.zeros(PMI.shape[1])
623       if self._neg_stationary:
624         for _,v,p in self._gen.transition_probs(start=()): # stationary
              distribution
625           averages[idx[v]] += p
626 
627       for step0 in range(self._num_steps): # step0 = step - 1
628         if not self._neg_stationary:
629           averages = PMI[:,:,step0].mean(axis=0) # divide by mean over column
630         PMI[:,:,step0] /= averages[np.newaxis,:]
631 
632       self._PMI = PMI
633     self._negative = negative
634     for step0 in range(self._num_steps):
635       M = self._PMI[:, :, step0] / negative
636       mat = np.log(M.clip(1))
637       source_embedding, target_embedding = self.factor_matrix(mat, self.
            _dimension_per_step)
638       indices = slice(step0 * self._dimension_per_step, (step0 + 1) * self.
            _dimension_per_step)
639       # Table 1 in [GraRep] does not mention normalizing the individual
            embeddings before concatenating them together.
640       # However, https://github.com/ShelsonCao/GraRep scales each embedding
            by its L2 norm.
641       if normalize:
642         source_embedding = source_embedding / np.linalg.norm(source_embedding
              )
643         target_embedding = target_embedding / np.linalg.norm(target_embedding
              )
644       self._source_embedding[:, indices] = source_embedding
645       self._target_embedding[:, indices] = target_embedding
646 
647   def PMI(self, step: int = 1) -> pd.DataFrame:
648     return pd.DataFrame(self._PMI[:, :, step - 1], index=self.
          source_paths_str, columns=self.target_nodes_str)
649 
650   def _decode_raw(self, start: Tuple[Any,...], step: int = 1) -> pd.Series:
651     indices = slice((step - 1) * self._dimension_per_step, step * self.
          _dimension_per_step)
652     return np.exp(self._source_embedding[self._source_paths[start], indices]
          @ self._target_embedding[:,indices].T)
653 
654   @property
655   def config(self):
656     "get␣configuration"
657     cfg = dict(init_class=self.__class__.__name__, init_gen=self._gen._id,
          init_dimension=self._dimension, init_id=self._id,
658         init_num_steps=self._num_steps, init_pairwise=self._pairwise,
              init_neg_stationary=self._neg_stationary)
659     cfg.update(self._training_config)
660     return cfg
661 
662 class HON_Transition_Hierarchical_Embedding(ABCAsymmetricEmbedding):
663   """
664   Extends HON_NetMF_Embedding with window_size=1 and pairwise=False by adding
        a penalty term which ties the
665   embeddings together along the specified hierarchy.
666   Due to the penalty, we cannot use singular value decomposition (SVD) and
        had to rely on stochastic gradient
```

```
667   descent (SGD) instead - similar to gensim.models.Word2Vec.
668   We use skip-gram with negative sampling (SGNS) as optimization criterion
          and added a penalty proportional to
669   sum( norm(u-pu)**2  for u,pu in node_hierarchy.items() ).
670
671   The intuition for introducing the penalty stems from the observation, that
          the probability of a path
672   (e.g. P[A->B->C]) can be calculated from the sums of probabilities of
          longer paths (e.g. sum_x P[x->A->B->C])
673   - assuming stationarity or introducing a 'begin of path'-symbol.
674   Hence, the transition probabilities for some start path correspond to
          averages of transition probabilities
675   for longer paths.
676   While for short paths, there is enough data (i.e. high support for the
          estimated transition probabilities),
677   longer paths may benefit from being tied towards their parent path (i.e.
          path[1:]).
678   The notion of "short" and "longer" is controlled by the parameter
          min_length of calc_node_hierarchy.
679
680   Note that, gensim.models.Word2Vec tries very hard to speed calculation up,
          as explained by a blog article of
681   its author. Unfortunately, this implementation will be much slower.
682   """
683   @staticmethod
684   def calc_node_hierarchy(gen: ABCHigherOrderPathGenerator, min_length:int=1)
           -> Dict[Tuple[Any,...],Tuple[Any,...]]:
685     """
686     Calculates the default node_hierarchy, which is used in __init__(...,
            node_hierarchy='calc').
687
688     min_length specifies the minimal length of those paths, which are
            affected by the penalty.
689     """
690     assert min_length > 0, 'min_length␣must␣be␣>␣1'
691     return {key: key[1:] for key in gen.rule_keys if len(key) >= min_length +
            1 }
692
693   def __init__(self, gen: ABCHigherOrderPathGenerator, dimension: int=128,
694       node_hierarchy:Union[str,Dict[Tuple[Any,...],Tuple[Any,...]]]='calc',
            seed=None, neg_stationary:bool=True):
695     self._neg_stationary = neg_stationary # distribution for negative samples
            : stationary (True) or uniform (False)
696     source_paths = gen.source_paths
697     target_nodes = gen.target_nodes
698     super().__init__(gen, dimension, source_paths, target_nodes)
699     self._id = 'Hierarchical_GF' # todo
700     self.reset(seed)
701     if node_hierarchy == 'calc':
702       node_hierarchy = self.calc_node_hierarchy(gen)
703     assert isinstance(node_hierarchy, collections.abc.Mapping), "
            node_hierarchy␣must␣be␣either␣the␣string␣'calc'␣or␣a␣dictionary␣with
            ␣keys␣containing␣paths␣and␣values␣containing␣the␣parent␣of␣each␣path
            ,␣i.e.␣key[1:]"
704     self._parent = node_hierarchy
705     children = collections.defaultdict(set)
706     for u,pu in node_hierarchy.items():
707       children[pu].add(u)
708     self._children = children
709
710   def clone(self):
711     "Clones␣the␣instance"
712     res = HON_Transition_Hierarchical_Embedding(self._gen, self._dimension,
            self._parent, None, self._neg_stationary)
713     res._total_steps = self._total_steps
714     res._source_embedding = self._source_embedding.copy()
715     res._target_embedding = self._target_embedding.copy()
716     res._rng.bit_generator.state = self._rng.bit_generator.state
717     res._training_history = list(self._training_history)
718     return res
719
720   def reset(self, seed=None):
721     "Resets␣the␣embedding␣(matrices␣are␣initialized␣with␣random␣values)."
722     dimension = self._dimension
```

```
723     self._seed = seed
724     self._rng = np.random.default_rng(seed)
725     self._total_steps = 0
726     self._source_embedding = self._rng.normal(0, 1/dimension, (len(self.
            _source_paths),dimension))
727     self._target_embedding = self._rng.normal(0, 1/dimension, (len(self.
            _target_nodes),dimension))
728     self._training_history = list()
729 
730   @staticmethod
731   def sigmoid(x: float) -> float: # better use a lookup table
732     # https://stackoverflow.com/questions/3985619/how-to-calculate-a-logistic
            -sigmoid-function-in-python ... ore use lookup table
733     # sigmoid(-710) raises a math range error
734     return  1 / (1  + math.exp(-max(x,-700)))
735 
736   def _calc_objective(self) -> Tuple[float,float,float]:
737     "calculates␣the␣loss␣function␣(SGNS␣and␣penalty␣are␣returned␣separately)"
738     loss = 0
739     loss_neg = 0
740     # neg_probs = distribution of negative samples
741     if self._neg_stationary: # stationary distribution
742       neg_probs = { next_node:prob for _,next_node,prob in self._gen.
              transition_probs(start=()) }
743     else: # uniform distribution
744       n_target = len(self._target_nodes)
745       neg_probs = { next_node:1/n_target for next_node in self._target_nodes
              }
746     for u,iu in self._source_paths.items(): # uniform sampling
747       probs = { next_node:prob for _,next_node,prob in self._gen.
              transition_probs(u) }
748       for v,iv in self._target_nodes.items():
749         score = self._source_embedding[iu,:] @ self._target_embedding[iv,:]
750         sigmoid_value = self.sigmoid(score)
751         loss -= math.log(sigmoid_value) * probs.get(v,0)
752         loss_neg -= math.log(1-sigmoid_value) * neg_probs[v]
753     loss_penalty = 0
754     for u,pu in self._parent.items():
755       iu = self._source_paths[u]
756       ipu = self._source_paths[pu]
757       loss_penalty += np.linalg.norm(self._source_embedding[iu,:] - self.
              _source_embedding[ipu,:])**2
758     return (loss, loss_neg, loss_penalty) # objective is loss + negative *
            loss_neg + penalty * loss_penalty
759 
760   def _update(self, iu, iv, label, learning_rate):
761     "SGD␣update"
762     eu = self._source_embedding[iu,:]
763     ev = self._target_embedding[iv,:]
764     score = eu @ ev
765     grad = learning_rate * (self.sigmoid(score) - label)
766     self._source_embedding[iu,:] -= grad * ev
767     self._target_embedding[iv,:] -= grad * eu
768 
769   def _update_hierarchy(self, u, iu, learning_rate, penalty, max_start_len:
          int):
770     "Calculates␣the␣gradient␣of␣the␣penalty␣and␣updates␣the␣source␣embedding␣
            accordingly."
771     factor = min(2 * learning_rate * penalty, 1) # avoid overshooting, which
            could result in a "math domain error" in the sigmoid calculation
772     if u in self._children:
773       if len(u) == max_start_len:
774         return
775       ichildren = list(self._source_paths[cu] for cu in self._children[u])
776       ec = self._source_embedding[ichildren,:].mean(axis=0) # average
              embedding for all children
777       self._source_embedding[iu,:] += factor * (ec - self._source_embedding[
              iu,:])
778 
779     pu = self._parent.get(u, None)
780     if not pu is None:
781       ipu = self._source_paths[pu]
782       self._source_embedding[iu,:] += factor * (self._source_embedding[ipu,:]
              - self._source_embedding[iu,:])
```

```
783
784   def train(self, steps: int=1, negative: int=1, penalty: float=0,
785       learning_rate_start: float=0.0025, learning_rate_end: Optional[float]=
              None,
786       max_start_len: Optional[int]=None, debug_objective: Optional[int]=None)
              :
787     """
788     Trains the embeddings using SGD.
789
790     Parameters
791     ----------
792
793     steps : int, >0
794
795       The number of SGD updates (not counting negative samples) per each path
              in source_paths.
796
797     negative : int, >= 0
798
799       Number of negative samples.
800
801     penalty: float, >= 0
802
803       Factor for the penalty terms.
804
805     learning_rate_start: float, >0
806
807       Learning rate. Linearly decreasing learning rates are specified with
              learning_rate_end != None.
808
809     learning_rate_end: float, optional (default=None)
810
811       Allows for linearly falling learning rates.
812
813     max_start_len: int, optional (default = None)
814
815       Allows for training source embeddings corresponding to short
              source_paths only.
816       At the end, the remaining source embeddings are replaced by their
              corresponding ancestors.
817       (max_start_len = None disables this feature.)
818       Source embeddings for a path longer than max_start_len are replaced by
              the ones corresponding
819       to shorter ancestors (i.e. path[-max_start_len:]).
820
821     debug_objective: int, optional (default = None)
822
823       If not None, this method returns the objective function evaluated in
              periodic intervals.
824       The length of the interval is specified by the parameter
              debug_objective.
825     """
826     self._training_history.append(dict(steps=steps, negative=negative,
          penalty=penalty,
827       learning_rate_start=learning_rate_start, learning_rate_end=
              learning_rate_end,
828       max_start_len=max_start_len)) # skip debug_objective
829     list_of_objectives = []
830     list_of_objectives_pos = []
831     list_of_objectives_neg = []
832     list_of_objectives_penalty = []
833     learning_rate_delta = 0 if learning_rate_end is None else
          learning_rate_end - learning_rate_start
834     if max_start_len is None:
835       source_paths_s = list((u,iu) for u,iu in self._source_paths.items())
836       max_start_len_eff = max(len(key) for key in source_paths_s)
837     else:
838       source_paths_s = list((u,iu) for u,iu in self._source_paths.items() if
              len(u) <= max_start_len)
839       max_start_len_eff = max_start_len
840     def copy_long_key_embeddings():
841       if max_start_len is not None:
842         # copy embeddings corresponding to keys longer than max_start_len
843         for u,iu in self._source_paths.items():
```

```
844             if len(u) > max_start_len:
845               pu = u[(-max_start_len_eff):] # FIXME: hardcoded assumption about
                       hode_hierarchy!
846               ipu = self._source_paths[pu]
847               self._source_embedding[iu,:] = self._source_embedding[ipu,:]
848
849       n_to = len(self._target_nodes)
850       self._rng.shuffle(source_paths_s)
851       i = 0
852       imax = steps * len(source_paths_s) - 1
853       for step in range(steps):
854         if debug_objective is not None:
855           copy_long_key_embeddings()
856           if step % int(debug_objective) == 0: # calculate objective every {
                    debug_objective} step
857             loss, loss_neg, penalty_loss = self._calc_objective()
858             list_of_objectives.append(loss + negative * loss_neg + penalty *
                    penalty_loss)
859             list_of_objectives_pos.append(loss)
860             list_of_objectives_neg.append(loss_neg)
861             list_of_objectives_penalty.append(penalty_loss)
862         for u,iu in source_paths_s: # uniform sampling
863           learning_rate = learning_rate_start + learning_rate_delta * (i/imax)
864           v = self._gen.random_step(start=u, rng=self._rng)
865           iv = self._target_nodes[v]
866           self._update(iu, iv, 1, learning_rate) # 1-sigma(x) = sigma(-x)
867           if self._neg_stationary: # stationary distribution
868             for v_neg in self._gen.random_step(start=(), size=negative, rng=
                      self._rng):
869               iv_neg = self._target_nodes[v_neg]
870               self._update(iu, iv_neg, 0, learning_rate) # 0-sigma(x) = -sigma(
                        x)
871           else: # uniform distribution
872             for iv_neg in self._rng.integers(n_to, size=negative):
873               self._update(iu, iv_neg, 0, learning_rate) # 0-sigma(x) = -sigma(
                        x)
874           if penalty > 0:
875             self._update_hierarchy(u, iu, learning_rate, penalty, max_start_len
                    )
876           i+=1
877       copy_long_key_embeddings()
878       if debug_objective is not None:
879         # finally, evaluate the objective once more.
880         loss, loss_neg, penalty_loss = self._calc_objective()
881         list_of_objectives.append(loss + negative * loss_neg + penalty *
                penalty_loss)
882         list_of_objectives_pos.append(loss)
883         list_of_objectives_neg.append(loss_neg)
884         list_of_objectives_penalty.append(penalty_loss)
885       self._total_steps += steps
886       if debug_objective is not None:
887         return dict(total_steps=self._total_steps, steps=steps, penalty=penalty
                ,
888           negative=negative, neg_stationary=self._neg_stationary,
889           learning_rate_start=learning_rate_start, learning_rate_end=
                learning_rate_start+learning_rate_delta,
890           dimension=self.dimension, max_start_len=max_start_len, objectives=np.
                array(list_of_objectives),
891           objectives_pos=np.array(list_of_objectives_pos), objectives_neg=np.
                array(list_of_objectives_neg),
892           objectives_penalty=np.array(list_of_objectives_penalty))
893
894     @property
895     def config(self):
896       "get␣configuration"
897       cfg = dict(init_class=self.__class__.__name__, init_gen=self._gen._id,
                init_dimension=self._dimension, init_id=self._id,
898         init_seed=self._seed, init_neg_stationary=self._neg_stationary)
899       # node_hierarchy is missing
900       for i,th in enumerate(self._training_history):
901         cfg.update({ f'train{i}_{k}':v for k,v in th.items() })
902       return cfg
```

C.6 Visualizations.py

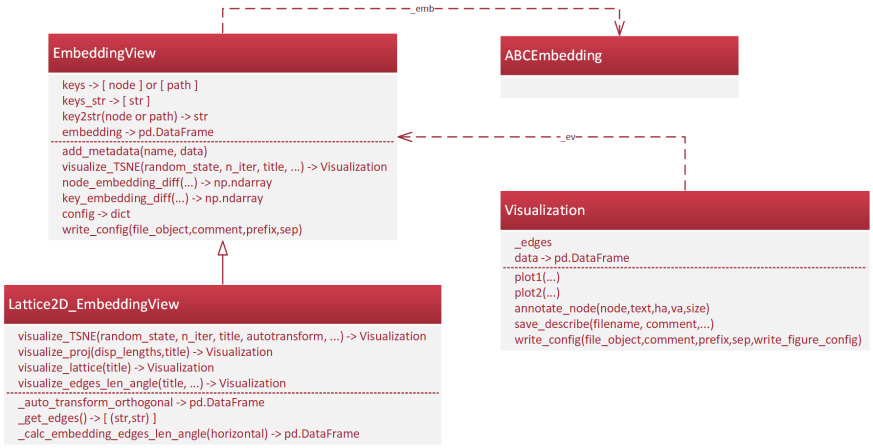


Figure C.3: UML class diagram for Visualizations.py

Due to the decision to implement also 'mixed' interactions (see § 3.1.2), a word or context may refer to a single node or a tuple of nodes, for which we use the umbrella term *key* — as already mentioned in § C.1, paragraph embedding. All embedding implementations provide either one or two pandas DataFrames whose rows contain the individual embeddings. How do we locate the embedding corresponding to a given node? First, determine the corresponding key (either the node itself or a tuple containing only the node). Second, we convert the key into a string. This process is simply

$$node \overset{node2key}{\longrightarrow} key \overset{key2str}{\longrightarrow} key\_str,$$

but it is tedious to distinguish, e.g., whether the list of keys equals nodes, source_paths, or target_nodes. To obtain a common interface for the different cases, we use the `EmbeddingView` as an adapter, whose primary task is to provide data (i.e., transformation by t–SNE) for the `Visualization`. The latter provides two plot methods, although one was basically obsoleted by adding grid lines to the plot. For a HON Lattice 2D, we use the derived class `Lattice2D_EmbeddingView` instead.

```
1  from typing import Any, Callable, Dict, List, Iterator, Tuple, Optional
2  #from SyntheticNetworks import *
3  from Embedding import ABCEmbedding, ABCSymmetricEmbedding
4  from sklearn.manifold import TSNE
5  import matplotlib.pyplot as plt
6  from matplotlib.collections import LineCollection
7  import seaborn as sns
8  import numpy as np
9  import pandas as pd
10 import math
11 import os
12
13 class Visualization(object):
14   def __init__(self, ev: "EmbeddingView", title, data, cols, config=dict(),
         edges=None):
15     self._ev = ev
16     self._title = title
```

```
17      self._data = data
18      self._cols = cols
19      self.config = config
20      self._edges = edges
21      self._edge_args = dict(color='gray', linewidth=1, linestyle=':')
22      self._figure = None
23      self._figure_config = {}
24
25    def __repr__(self):
26      othercols = sorted(list(set(self._data.columns).difference(self._cols)))
27      return f"""paths: {self.paths_id}
28  embedding: {self.emb_id}
29  dim: {self._data.shape}
30  embedding columns: {str(self._cols)}
31  other columns: {str(othercols)}"""
32
33    @property
34    def paths_id(self) -> str:
35      "ID␣of␣the␣paths␣(transition␣probabilities)"
36      return self._ev._emb._gen._id
37
38    @property
39    def emb_id(self) -> str:
40      "ID␣of␣the␣embedding"
41      return self._ev._emb._id + ('_source' if self._ev.use_source else '')
42
43    @property
44    def data(self) -> pd.DataFrame:
45      "returns␣a␣DataFrame␣containing␣the␣(visualization␣of␣the)␣embedding␣
            togehter␣with␣explaining␣attributes."
46      return self._data
47
48    def _add_edges(self, ax, data, **kwargs):
49      if self._edges is None:
50        return
51      data_xy = data[self._cols]
52      lines = list([data_xy.loc[n1], data_xy.loc[n2]] for n1,n2 in self._edges)
53      lc = LineCollection(lines, **kwargs)
54      ax.add_collection(lc)
55
56    def plot1(self, figsize: Tuple[float,float]=(6,4), dpi: int=200, figureargs
          =dict(),
57          filter_col: Optional[str]=None, filter_values=set(), rotate:float=0,
58          return_figure=False, **kwargs):
59      """
60      Displays a visualization by a seaborn.scatterplot
61
62      Parameters
63      ----------
64      figsize : (float,float)
65        Width, height in inches.
66
67      dpi : int, default=200
68        Resolution of the figure.
69
70      figureargs : dict
71        These are passed to pyplot.figure
72
73      filter_col : str, optional
74        If specified, the data is filtered by this column, see filter_values.
75
76      filter_values : set
77        If filter_column is specified, consider only records where the column
              filter_col has values in filter_values.
78
79      return_figure: bool, default=False
80        If True, the figure is returned.
81
82      kwargs :
83        These are passed to seaborn.scatterplot (e.g. hue='col1', style='col2',
              alpha=0.5, palette='coolwarm').
84      """
85      data = self._data
86      x_name, y_name = self._cols
```

```
87      if rotate != 0: # rotate before edges
88        data = data.copy()
89        c,s = math.cos(rotate), math.sin(rotate)
90        x = c*data[x_name] + s*data[y_name]
91        y = -s*data[x_name] + c*data[y_name]
92        data[x_name] = x
93        data[y_name] = y
94      fig = plt.figure(figsize=figsize, dpi=dpi, **figureargs)
95      if self._title != '':
96        fig.suptitle(self._title)
97      self._add_edges(fig.gca(), data=data, **self._edge_args)
98      palette = kwargs.pop('palette', 'coolwarm')
99      if filter_col is not None: # apply filter after edges
100       data = data[data[filter_col].isin(filter_values)]
101     ax = sns.scatterplot(data=data, x=x_name, y=y_name, palette=palette, ax=
            fig.gca(), **kwargs)
102     ax.set_aspect(1)
103     # caution: by default there nothing to display in legend; should check
            for hue, style, ... in kwargs.
104     ax.legend(loc='center␣left', bbox_to_anchor=(1, 0.5), ncol=1)
105     self._figure = fig
106     self._figure_config = dict(figsize=figsize, dpi=dpi, figureargs=repr(
            figureargs),
107         filter_col=filter_col, filter_values=repr(filter_values), rotate=
                rotate, **kwargs)
108     if return_figure:
109       return fig
110
111   def plot2(self, figsize: Tuple[float,float]=(15,10), dpi: int=200,
          figureargs=dict(),
112       filter_col: Optional[str]=None, filter_values=set(),
113       return_figure=False, **kwargs):
114     """
115     Displays a visualization by two seaborn.scatterplots.
116     The columns x_orig and y_orig added by Lattice2D_EmbeddingView.
            _add_lattice_coord() are displayed as hue.
117
118     Parameters
119     ----------
120     figsize : (float,float)
121       Width, height in inches.
122
123     dpi : int, default=200
124       Resolution of the figure.
125
126     figureargs : dict
127
128     filter_col : str, optional
129       If specified, the data is filtered by this column, see filter_values.
130
131     filter_values : set
132       If filter_column is specified, consider only records where the column
              filter_col has values in filter_values.
133
134     return_figure: bool, default=False
135       If True, the figure is returned.
136
137     kwargs :
138       These are passed to seaborn.scatterplot (e.g. style='key_len', alpha
              =0.5, palette='coolwarm').
139     """
140     data = self._data
141     x_name, y_name = self._cols
142
143     constrained_layout=figureargs.pop('constrained_layout',True)
144     fig, axs = plt.subplots(1,2, figsize=figsize, dpi=dpi, **figureargs,
            constrained_layout=constrained_layout)
145     if self._title != '':
146       fig.suptitle(self._title)
147     self._add_edges(axs[0], data, **self._edge_args)
148     self._add_edges(axs[1], data, **self._edge_args)
149
150     if filter_col is not None: # apply filter after edges
151       data = data[data[filter_col].isin(filter_values)]
```

```
152
153     palette = kwargs.pop('palette', 'coolwarm')
154     scatterplot_args = dict(data=data, x=x_name, y=y_name, palette=palette,
            **kwargs)
155
156     ax = sns.scatterplot(hue='x_orig', ax=axs[0], **scatterplot_args)
157     ax.set_aspect(1)
158     ax.legend(loc='center left', bbox_to_anchor=(1, 0.5), ncol=1)
159
160     ax = sns.scatterplot(hue='y_orig', ax=axs[1], **scatterplot_args)
161     ax.set_aspect(1)
162     ax.legend(loc='center left', bbox_to_anchor=(1, 0.5), ncol=1)
163     self._figure = fig
164     self._figure_config = dict(figsize=figsize, dpi=dpi, figureargs=repr(
            figureargs),
165         filter_col=filter_col, filter_values=repr(filter_values), **kwargs)
166     if return_figure:
167       return fig
168
169   def annotate_node(self, node, text=None, ha='center', va='center', size=8):
170     # ha in { 'left', 'center', 'right' }, va in { 'top', 'center', 'bottom'
            }
171     ev = self._ev
172     coord = self.data.loc[ev.key2str(ev.node2key(node)), self._cols]
173     if text is None:
174       text = str(node)
175     for ax in self._figure.axes:
176       ax.annotate(text, coord, horizontalalignment=ha, verticalalignment=va,
              size=size)
177
178   def write_config(self, file_object, comment: str = '', prefix: str = '',
          sep: str ='\t', write_figure_config: bool = True):
179     self._ev.write_config(file_object=file_object, comment=comment, prefix=
            prefix, sep=sep)
180     file_object.write(prefix + '\n' + prefix + 'Visualization:\n')
181     for k,v in self.config.items():
182       file_object.write(prefix + '%s%s%s\n' % (k,sep,v))
183     if write_figure_config:
184       file_object.write(prefix + '\n' + prefix + 'Plot:\n')
185       for k,v in self._figure_config.items():
186         file_object.write(prefix + '%s%s%s\n' % (k,sep,v))
187
188   def save_describe(self, filename, comment : str = '', bbox_inches='tight',
          **kwargs):
189     "saves the most recent figure created by this instance and exports the
            configs to a text file."
190     self._figure.savefig(filename, bbox_inches=bbox_inches, **kwargs)
191     ev = self._ev
192     emb = ev._emb
193     gen = emb._gen
194     with open(filename + '.txt','w') as f:
195       self.write_config(f, comment=comment, prefix='', sep='\t',
              write_figure_config=True)
196
197 class EmbeddingView(object):
198   "Helper class for visualizing embeddings"
199   def __init__(self, emb : ABCEmbedding, use_source: bool = True):
200     self._emb = emb
201     self._series = dict()
202     if isinstance(emb, ABCSymmetricEmbedding):
203       self._keys = list(emb.nodes)
204       self._keys_str = emb.nodes_str
205       self.key2str = emb.key2str
206       use_source = False
207     elif use_source:
208       self._keys = list(emb.source_paths)
209       self._keys_str = emb.source_paths_str
210       self.key2str = emb.path2str
211     else:
212       self._keys = list(emb.target_nodes)
213       self._keys_str = emb.target_nodes_str
214       self.key2str = emb.node2str
215     self.use_source = use_source # True if key is a tuple of nodes
216     if use_source:
```

```
217       for name, data in emb._gen._source_path_metadata.items():
218         self.add_metadata(name, data)
219     else:
220       for name, data in emb._gen._target_node_metadata.items():
221         self.add_metadata(name, data)
222
223   def add_metadata(self, name, data):
224     # data maps the keys to some grouping criterion
225     self._series[name] = pd.Series({self.key2str(key): data[key] for key in
          self.keys})
226     return self
227
228   def __getitem__(self, series_name) -> pd.Series:
229     return self._series[series_name]
230
231   def _append_metadata(self, data: pd.DataFrame, copy=False) -> pd.DataFrame:
232     res = data.copy() if copy else data
233     for name, s in self._series.items():
234       res[name] = s
235     return res
236
237   def node2key(self, node:Any):
238     "converts␣a␣node␣into␣a␣key"
239     #return self._emb.node2key(node, self.use_source) # todo: remove
          ABCEmbedding.node2key?
240     return (node,) if self.use_source else node
241
242   @property
243   def keys(self): # -> List[Any]:
244     "Keys␣used␣for␣the␣embedding"
245     return self._keys
246
247   @property
248   def keys_str(self): # -> List[str]:
249     "String␣representation␣(i.e␣self.key2str)␣of␣the␣keys.␣(Equals␣embedding.
          index)"
250     return self._keys_str
251
252   @property
253   def embedding(self) -> pd.DataFrame:
254     "Embedding,␣index␣corresponds␣to␣keys_str␣and␣columns␣are␣the␣dimensions␣
          o␣the␣embedding␣space."
255     if isinstance(self._emb, ABCSymmetricEmbedding):
256       return self._emb.embedding
257     elif self.use_source:
258       return self._emb.source_embedding
259     else:
260       return self._emb.target_embedding
261
262   def node_embedding_diff(self, node_pairs) -> np.ndarray:
263     "For␣a␣given␣list␣of␣first␣oder␣edges␣(pairs␣of␣nodes),␣determine␣the␣
          difference␣in␣the␣embedding␣space␣between␣them"
264     key_pairs = list((self.node2key(v), self.node2key(w)) for v,w in
          node_pairs)
265     return self.key_embedding_diff(key_pairs)
266
267   def key_embedding_diff(self, key_pairs) -> np.ndarray:
268     "For␣a␣given␣list␣of␣pairs␣of␣keys,␣determine␣the␣difference␣in␣the␣
          embedding␣space␣between␣them"
269     tmp = np.zeros(shape=(len(key_pairs), self._emb.dimension))
270     i=0
271     for start,end in key_pairs:
272       e_start = self._emb[start] # = self.embedding.loc[key2str(start)]
273       e_end = self._emb[end] # = self.embedding.loc[key2str(end)]
274       tmp[i,:] = e_end - e_start
275       i+=1
276     return tmp
277
278   def visualize_TSNE(self, random_state=1, n_iter=1000, title: Optional[str]=
        None, **kwargs) -> Visualization:
279     "Returns␣TSNE␣visualization"
280     config = dict(creator='visualize_TSNE', random_state=random_state, n_iter
          =n_iter, title=title, **kwargs)
281     embedding = self.embedding
```

```
282     cols = ['x','y']
283     data = TSNE(n_components=2, random_state=random_state, n_iter=n_iter, **
            kwargs).fit_transform(embedding)
284     data = pd.DataFrame(data, index=embedding.index, columns=cols)
285     if title is None:
286       title = 'TSNE(random_state=%d)' % random_state
287     return Visualization(self, title, self._append_metadata(data), cols,
            config)
288
289   @property
290   def config(self):
291     "get␣configuration"
292     return dict(init_class=self.__class__.__name__, use_source=self.
            use_source)
293
294   def write_config(self, file_object, comment: str = '', prefix: str = '',
          sep: str ='\t'):
295     self._emb.write_config(file_object=file_object, comment=comment, prefix=
            prefix, sep=sep)
296     file_object.write(prefix + '\n' + prefix + 'EmbeddingView:\n')
297     for k,v in self.config.items():
298       file_object.write(prefix + '%s%s%s\n' % (k,sep,v))
299
300 class Lattice2D_EmbeddingView(EmbeddingView):
301   "EmbeddingView␣for␣the␣synthetic␣network␣'Lattice2D_2nd_order_dynamic'"
302   def __init__(self, emb : ABCEmbedding, use_source: bool = True,
          edge_distance: Optional[int] = 1):
303     super().__init__(emb, use_source)
304     self.creator = emb._gen.creator # todo: check has_attr and type
305     self._edge_distance = edge_distance
306     if edge_distance is None:
307       self._hor_edge_pairs = []
308       self._ver_edge_pairs = []
309     elif edge_distance == 1:
310       self._hor_edge_pairs = self.creator.horizontal_edges1
311       self._ver_edge_pairs = self.creator.vertical_edges1
312     else:
313       self._hor_edge_pairs = list((k1[0],k2[1]) for k1,k2 in self.creator.
              horizontal_edges2)
314       self._ver_edge_pairs = list((k1[0],k2[1]) for k1,k2 in self.creator.
              vertical_edges2)
315
316   def visualize_proj(self, disp_lengths: bool=True, title: Optional[str]=None
          ) -> Visualization:
317     """
318     Returns a projection visualization.
319     The projection is determined utilizing knowledge about lattice structure
            (i.e. 'right' and 'down' neighbors).
320     """
321     config = dict(creator='visualize_proj', disp_lengths=disp_lengths, title=
            title)
322     # Since TSNE may distort the data, find average directions of horizontal
            and vertical edges
323     max_key_len = max(len(key) for key in self._keys) if self.use_source else
             1
324     if max_key_len == 1:
325       hor_edge_diffs = self.node_embedding_diff(self.creator.
              horizontal_edges1)
326       ver_edge_diffs = self.node_embedding_diff(self.creator.vertical_edges1)
327     else:
328       hor_edge_diffs = self.key_embedding_diff(self.creator.horizontal_edges2
              )
329       ver_edge_diffs = self.key_embedding_diff(self.creator.vertical_edges2)
330     if disp_lengths:
331       hor_edge_lengths = np.linalg.norm(hor_edge_diffs, axis=1)
332       ver_edge_lengths = np.linalg.norm(ver_edge_diffs, axis=1)
333       print('max_key_len', max_key_len)
334       print('average␣horizontal␣edge␣length', hor_edge_lengths.mean())
335       print('average␣vertical␣edge␣length', ver_edge_lengths.mean())
336       print('ratio', hor_edge_lengths.mean()/ver_edge_lengths.mean())
337     mean_hor_edge = hor_edge_diffs.mean(axis=0)
338     mean_ver_edge = ver_edge_diffs.mean(axis=0)
339     embedding = self.embedding
340     cols = ['proj_hor','proj_ver']
```

```
341       emb_proj_hor = embedding @ (mean_hor_edge / np.linalg.norm(mean_hor_edge)
              )
342       emb_proj_ver = embedding @ (mean_ver_edge / np.linalg.norm(mean_ver_edge)
              )
343       data = pd.DataFrame(np.array([emb_proj_hor, emb_proj_ver]).T, index=
              embedding.index, columns=cols)
344       if title is None:
345         title = 'Projection␣along␣average␣horizontal␣&␣vertical␣edge'
346       edges = self._get_edges()
347       return Visualization(self, title, self._append_metadata(data), cols,
              config, edges)
348
349     def visualize_lattice(self, title: Optional[str]=None) ->  Visualization:
350       "Displays␣the␣coordinates␣of␣the␣embeddings"
351       config = dict(creator='visualize_lattice', title=title)
352       data = pd.DataFrame({'x': self._series['x_orig'], 'y': self._series['
              y_orig']})
353       cols = data.columns.tolist()
354       edges = self._get_edges()
355       return Visualization(self, title, self._append_metadata(data), cols,
              config, edges)
356
357     def _auto_transform_orthogonal(self, vis_data, verbose=False) -> pd.
          DataFrame:
358       """
359       Visualizations are often defined in terms of relative distances,
360       which are not affected by orthogonal transformations.
361       Givent the ground truth is known, let us try to align the visualization.
362       """
363       # We would like to use horizontal/vertical edges to determione the
              direction of the x and y axis.
364       # The edges should be oriented from left to right and from down to up.
365       # However, some embeddings separate the nodes into two clusters (even and
               odd).
366       # Therefore, take horizontal and vertical edges of distance 2 unless
              specified otherwise.
367       def vis_diff(vis_data, node_pairs):
368         tmp = np.zeros(shape=(len(node_pairs), vis_data.shape[1]))
369         i=0
370         for start,end in node_pairs:
371           v_start = vis_data.loc[self.key2str(self.node2key(start))]
372           v_end = vis_data.loc[self.key2str(self.node2key(end))]
373           tmp[i,:] = v_end - v_start
374           i += 1
375         return tmp
376       hor_edge_diffs = vis_diff(vis_data, self._hor_edge_pairs)
377       ver_edge_diffs = vis_diff(vis_data, self._ver_edge_pairs)
378       hor_edge_mean = hor_edge_diffs.mean(axis=0)
379       ver_edge_mean = ver_edge_diffs.mean(axis=0)
380       det = np.linalg.det(np.matrix([hor_edge_mean,ver_edge_mean]))
381       if verbose:
382         print(hor_edge_mean, ver_edge_mean, det)
383       if det < 0:
384         y_col = vis_data.columns[1]
385         vis_data = vis_data.copy()
386         vis_data[y_col] = -vis_data[y_col]
387         hor_edge_mean[1] *= -1
388         ver_edge_mean[1] *= -1
389       # determine the angle by which hor_edge_mean should be rotated to point
              towards [1,0].
390       angle1 = -math.atan2(hor_edge_mean[1], hor_edge_mean[0])
391       c, s = math.cos(angle1), math.sin(angle1)
392       rot = np.matrix([[c,-s],[s,c]])
393       if verbose:
394         print('1st␣rotation␣by', angle1, 'results␣in', (rot @ hor_edge_mean).A1
              , (rot @ ver_edge_mean).A1)
395
396       # Determine the angle by which ver_edge_mean should be rotated to point
              towards [0,1].
397       # To avoid calculations modulo 2*pi estimate the angle2 after applying
              the rotation by angle1.
398       # Checking the determinant already ensures that angle2 is in [-pi/2, pi
              /2], and I expect angle2 to be small.
399       rot_ver_edge_mean = (rot @ ver_edge_mean).A1
```

```
400     angle2 = math.atan2(rot_ver_edge_mean[0], rot_ver_edge_mean[1])
401     if verbose:
402       print('angle2', angle2)
403     angle = angle1 + angle2/2 # take average
404     c,s = math.cos(angle),math.sin(angle)
405     rot = np.matrix([[c,-s],[s,c]])
406     if verbose:
407       print('2nd rotation by', angle, 'results in', (rot @ hor_edge_mean).A1,
              (rot @ ver_edge_mean).A1)
408     return pd.DataFrame(vis_data.values @ rot.T, index=vis_data.index,
            columns=vis_data.columns)
409
410   def _get_edges(self):
411     if self._edge_distance is None:
412       return None
413     edges = self._hor_edge_pairs + self._ver_edge_pairs
414     # convert pairs of nodes into pairs of string representations of keys
415     return list((self.key2str(self.node2key(n1)),self.key2str(self.node2key(
            n2))) for n1,n2 in edges)
416
417   def visualize_TSNE(self, random_state=1, n_iter=1000, title: Optional[str]=
            None,
418       autotransform: bool = True, autotransform_verbose:bool = False, **
              kwargs) -> Visualization:
419     "Returns TSNE visualization"
420     config = dict(creator='visualize_TSNE', random_state=random_state, n_iter
            =n_iter, title=title,
421         autotransform=autotransform, **kwargs)
422     embedding = self.embedding
423     cols = ['x','y']
424     data = TSNE(n_components=2, random_state=random_state, n_iter=n_iter, **
            kwargs).fit_transform(embedding)
425     data = pd.DataFrame(data, index=embedding.index, columns=cols)
426     if title is None:
427       title = 'TSNE(random_state=%d)' % random_state
428     if autotransform:
429       data = self._auto_transform_orthogonal(data, verbose=
              autotransform_verbose)
430     edges = self._get_edges()
431     return Visualization(self, title, self._append_metadata(data), cols,
            config, edges)
432
433
434   def _calc_embedding_edges_len_angle(self, horizontal: bool):
435     "Calculates lengths and angles (to their average) of embedded edges"
436     edges = self._hor_edge_pairs if horizontal else self._ver_edge_pairs
437     keys = list(f'{n1} -> {n2}' for  n1,n2 in edges)
438     emb_edges = self.node_embedding_diff(edges)
439     mean_edge = emb_edges.mean(axis=0)
440     mean_edge1 = mean_edge / np.linalg.norm(mean_edge)
441     edges_len = np.linalg.norm(emb_edges,axis=1)
442     edges_spr = (emb_edges @ mean_edge1) / edges_len
443     edges_angle = np.arccos( edges_spr.clip(-1,1) )
444     return pd.DataFrame({'len': edges_len, 'angle': edges_angle, 'angle360':
            edges_angle *180/math.pi,
445         'direction': ('horizontal' if horizontal else 'vertical')}, index=
              keys)
446
447   def visualize_edges_len_angle(self, title: Optional[str]=None,
448       figsize: Tuple[float,float]=(6,4), dpi: int=200, figureargs=dict(), **
              kwargs):
449     config=dict(creator='visualize_edges_len_angle')
450     hor_stats = self._calc_embedding_edges_len_angle(True)
451     ver_stats = self._calc_embedding_edges_len_angle(False)
452     stats = hor_stats.append(ver_stats)
453     stats.sort_values('len', inplace=True) # in case of overplotting, treat
            horizontal and vertical edges equally
454     cols = ['len','angle']
455     if title is None:
456       title = 'Edge lengths and angles'
457     # This plot serves only to illustrate conceptual limitations of
            visualize_proj(),
458     # which does not justify changes in class Visualization.
459     # As a workaround, return an instance of Visualization, which has already
```

```
            created a figure.
460         vis = Visualization(self, title, stats, cols, config, edges=None)
461         vis.plot1(figsize=figsize, dpi=dpi, hue='direction', figureargs=
                figureargs, **kwargs)
462         ax = vis._figure.gca()
463         ax.set_aspect('auto')
464         ax.set_xlim(0, None)
465         ax.set_ylim(0, math.pi)
466         ax.tick_params(labelright=True, right=True)
467         ax.set_yticks(math.pi * np.linspace(0,1,num=5))
468         ax.set_yticklabels(['0',r'$\frac{\pi}{4}$',r'$\frac{\pi}{2}$',r'$\frac{3\
                pi}{4}$',r'$\pi$'])
469         ax.hlines(math.pi/2, *ax.get_xlim(),linewidths=0.5)
470         ax.legend(loc='upper left')
471         vis._figure_config['workaround'] = 'see visualize_edges_len_angle' # let
                save_describe() know about this workaround
472         return vis
473
474       @property
475       def config(self):
476         "get configuration"
477         cfg = super().config
478         cfg['edge_distance'] = self._edge_distance
479         if self._edge_distance is not None:
480           cfg['mean horizontal edge length'] = self.
                _calc_embedding_edges_len_angle(True)['len'].mean()
481           cfg['mean vertical edge length'] = self._calc_embedding_edges_len_angle
                (False)['len'].mean()
482           cfg['stretch'] = cfg['mean vertical edge length'] / cfg['mean
                horizontal edge length']
483         return cfg
484
485     ## Usage:
486     # from Visualizations import Lattice2D_EmbeddingView, EmbeddingView,
            Visualization
487     # from SyntheticNetworks import create_lattice_2nd_order_dynamic
488     # from Embedding import HON_DeepWalk_Embedding
489     # latgen = create_lattice_2nd_order_dynamic(size=10, omega=0.5, lattice_sep
            ='-')
490     # emb = HON_DeepWalk_Embedding(latgen, 128)
491     # %time emb.train(random_seed=1)
492     # ev = Lattice2D_EmbeddingView(emb)
493     # vis = ev.visualize_TSNE(random_state=1, n_iter=1000)
494     # vis.plot1(hue='even_odd', style='direction', rotate=2.4)
495     # vis.plot2(style='direction', alpha=0.5)
496     #
497     # only_len_1 = dict(filter_col='key_len', filter_values={1})
498     # only_len_2 = dict(filter_col='key_len', filter_values={2})
499     # vis.plot2(style='direction', alpha=0.5, **only_len_2)
```

C.7 Lattice2D_sim.ipynb

For the visualization experiment in § 4.1, we performed a grid search over the parameter space (figures 4.1 and 4.4). Below, we display a shortened variant of the corresponding jupyter notebook, `Lattice2D_sim (short).ipynb`. Other plots were created by `Plots_ExpVis_synth.ipynb` and `Plots_ExpVis_Word2vec.ipynb`, which we omit here, because the underlying functionality is part of the infractructure — with the exception of the concentration analysis of the PMI (figure 4.3) and the neighborhood matrix (figure 4.6, left).

In [1]:

```
comment = 'calculated in Lattice2D_sim (short).ipynb'
verbose = True
```

In [2]:

```
%matplotlib inline
#%matplotlib notebook
import numpy as np
import pandas as pd
import math
import os
import matplotlib.pyplot as plt
#from tqdm import tqdm
from tqdm.notebook import tqdm
```

In [3]:

```
from HigherOrderPathGenerator import CrossValidation_HigherOrderPathGenerator
from Embedding import HON_DeepWalk_Embedding, HON_Node2vec_Embedding, HONEM_Embedding,
HON_NetMF_Embedding, HON_GraRep_Embedding, HON_Transition_Hierarchical_Embedding
from SyntheticNetworks import create_lattice_2nd_order_dynamic
from Visualizations import Visualization, EmbeddingView, Lattice2D_EmbeddingView
```

In [4]:

```
size = 10
omega = 0.5
gen = create_lattice_2nd_order_dynamic(size, omega, lattice_sep='-', check=False)
```

In [5]:

```
if not os.path.exists('tmp'):
    os.makedirs('tmp')
```

In [6]:

```
def iter_cross_prod(**kwargs):
    #kwargs = { k:list(v) for k,v in kwargs.items() } #  generators to list
    total_len = np.prod([len(v) for v in kwargs.values()])
    for i in tqdm(range(total_len)):
        current = dict()
        #for k,v in kwargs.items(): # simpler, but wrong order
        #    current[k] = v[i % len(v)]
        #    i = i // len(v)
        j = total_len
        for k,v in kwargs.items():
            j = j // len(v) # = product of lengths over remaining items
            current[k] = v[i // j]
            i = i % j
        yield current
```

In [7]:

```
# verifying order of iteration
for current in iter_cross_prod(a=[1,2], b=[3,4,5], c=[6,7]):
    print(current)
```

```
{'a': 1, 'b': 3, 'c': 6}
{'a': 1, 'b': 3, 'c': 7}
{'a': 1, 'b': 4, 'c': 6}
{'a': 1, 'b': 4, 'c': 7}
{'a': 1, 'b': 5, 'c': 6}
{'a': 1, 'b': 5, 'c': 7}
{'a': 2, 'b': 3, 'c': 6}
{'a': 2, 'b': 3, 'c': 7}
{'a': 2, 'b': 4, 'c': 6}
{'a': 2, 'b': 4, 'c': 7}
{'a': 2, 'b': 5, 'c': 6}
{'a': 2, 'b': 5, 'c': 7}
```

In [8]:

```
def get_stretch_ratio(emb, edge_distance = 1):
    ev = Lattice2D_EmbeddingView(emb, use_source=True, edge_distance=edge_distance)
    hor_edges_len = np.linalg.norm(ev.node_embedding_diff(ev._hor_edge_pairs),axis=1)
    ver_edges_len = np.linalg.norm(ev.node_embedding_diff(ev._ver_edge_pairs),axis=1)
    return ver_edges_len.mean()/hor_edges_len.mean() # = ev.config['stretch']
```

In [9]:

```
def disp_stretch_W(df, emb=None, title=None, figsize=(12,3), dpi=200, bbox_inches='tigh
t', filename=None):
    df = df.sort_values('window_size')
    fig =plt.figure(figsize=figsize, dpi=dpi)
    if title is not None:
        fig.suptitle(title)
    ax = fig.gca()
    ax.plot(df['window_size'], df['stretch_ratio'])
    ax.set_xlabel('window_size')
    ax.set_ylabel('stretch_ratio')
    if filename is not None:
        fig.savefig(filename, bbox_inches=bbox_inches)
        with open(filename + '.txt','w') as f:
            if comment != '':
                f.write(comment + '\n\n')
            if emb is not None:
                emb.write_config(f, comment=comment, prefix='', sep='\t')
            f.write('\nData:\n')
            f.write(df[['window_size','stretch_ratio']].sort_values('stretch_ratio', as
cending=False).to_csv(index=False,line_terminator='\n'))
```

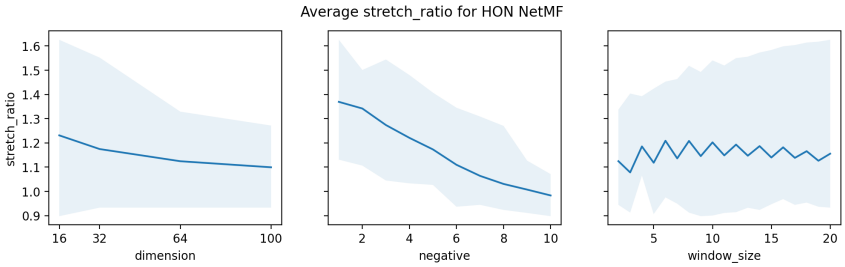
In [10]:

```
def disp_marginals(df, emb=None, title=None, figsize=(12,3), dpi=200, bbox_inches='tigh
t', filename=None, exclude_FON=True):
    "average stretch_ratio grouped by the values of each parameter"
    cols = sorted(set(df.columns) - {'stretch_ratio','name','q', 'p', 'num_walks'})
    p_vals = sorted(df['p'].unique()) if 'p' in df.columns else [None]
    if exclude_FON:
        if 'window_size' in df.columns:
            df = df[df['window_size']>1]
        if 'num_steps' in df.columns:
            df = df[df['num_steps']>1]
    fig,axs = plt.subplots(nrows=1, ncols=len(cols), sharey=True, figsize=figsize, dpi=
dpi)
    if title is not None:
        fig.suptitle(title)
    for c,ax in zip(cols, axs):
        for p in p_vals:
            df_p = df if p is None else df[df['p']==p]
            avgs = df_p.groupby(c)['stretch_ratio'].mean().sort_index()
            x_vals = avgs.index
            y_vals = avgs.values
            label = None if p is None else 'p='+str(p)
            ax.plot(x_vals, y_vals, label=label)
            if p is None:
                y_min = df_p.groupby(c)['stretch_ratio'].min().sort_index().values
                y_max = df_p.groupby(c)['stretch_ratio'].max().sort_index().values
                ax.fill_between(x_vals, y_min, y_max, alpha=0.1)
            ax.set_xlabel(c)
            if c == 'dimension' or c == 'factor':
                ax.set_xticks(x_vals)
    axs[0].set_ylabel('stretch_ratio')
    if len(p_vals) > 1:
        axs[0].legend()
    if filename is not None:
        fig.savefig(filename, bbox_inches=bbox_inches)
        with open(filename + '.txt','w') as f:
            if comment != '':
                f.write(comment + '\n\n')
            f.write('Params:\n')
            for c in cols:
                vals = sorted(df[c].unique())
                f.write(f'{c}\t{vals}\n')
            f.write('\nMarginals:\n')
            for c in cols:
                f.write(str(df.groupby(c)['stretch_ratio'].agg(['min','mean','max']))+'
\n')
            if emb is not None:
                emb.write_config(f, comment=comment, prefix='', sep='\t')
            f.write('\nTop 10:\n')
            df = df.sort_values('stretch_ratio', ascending=False)
            f.write(str(df.head(10))+'\n')
            f.write('\nBottom 10:\n')
            f.write(str(df.tail(10))+'\n')
```

NetMF

In [11]:

```
# NetMF
res_NetMF = []
emb = None
# loop over dimension and window_size first to allow NetMF to reuse its PMI calculatio
n.
for current in iter_cross_prod(dimension=[16, 32, 64, 100], window_size=range(1,21), ne
gative=range(1,11)):
    if (emb is None) or (emb._dimension!=current['dimension']):
        emb = HON_NetMF_Embedding(gen, dimension=current['dimension'], pairwise=True)
    emb.train(window_size=current['window_size'], negative=current['negative'])
    stetch = get_stretch_ratio(emb, edge_distance = 1)
    config = dict(stretch_ratio=stetch, name='NetMF')
    config.update(current)
    res_NetMF.append(config)
res_NetMF = pd.DataFrame(res_NetMF).sort_values('stretch_ratio', ascending=False)
disp_marginals(res_NetMF, emb, title='Average stretch_ratio for HON NetMF', filename='t
mp/stretch_sim_NetMF.png')
#res_NetMF.head(10)
```



In [12]:

```
res_NetMF[res_NetMF['window_size']==2].head()
```

Out[12]:

	stretch_ratio	name	dimension	window_size	negative
11	1.337149	NetMF	16	2	2
10	1.309763	NetMF	16	2	1
12	1.302841	NetMF	16	2	3
211	1.275984	NetMF	32	2	2
210	1.258308	NetMF	32	2	1

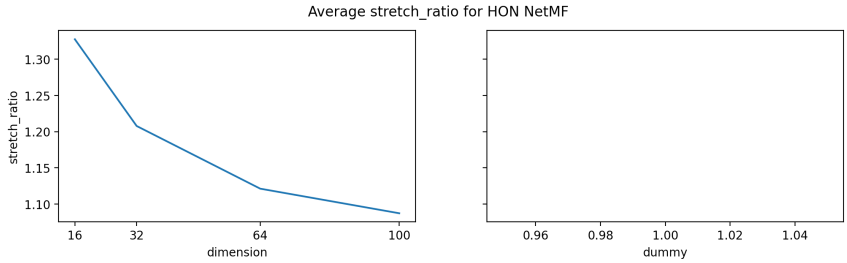
HONEM

In [13]:

```
# HONEM
res_HONEM = []
emb = None
# disp_marginals requires at least two parameters
for current in iter_cross_prod(dimension=[16, 32, 64, 100], dummy=[1]):
    emb = HONEM_Embedding(gen, dimension=current['dimension'])
    emb.train()
    stetch = get_stretch_ratio(emb)
    config = dict(stretch_ratio=stetch, name='HONEM')
    config.update(current)
    res_HONEM.append(config)
res_HONEM = pd.DataFrame(res_HONEM).sort_values('stretch_ratio', ascending=False)
disp_marginals(res_HONEM, emb, title='Average stretch_ratio for HON NetMF', filename='t
mp/stretch_sim_HONEM.png')
res_HONEM
```

Out[13]:

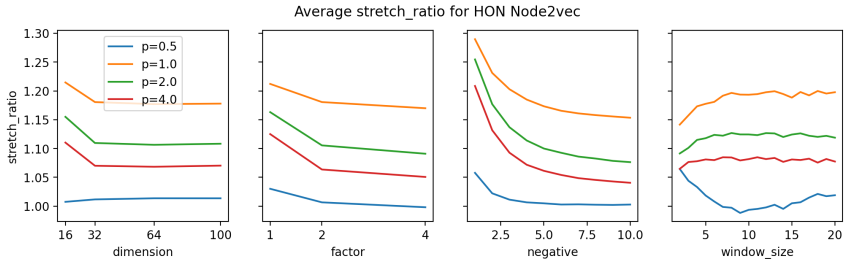
	stretch_ratio	name	dimension	dummy
0	1.327574	HONEM	16	1
1	1.207816	HONEM	32	1
2	1.121258	HONEM	64	1
3	1.087279	HONEM	100	1



DeepWalk & Node2vec

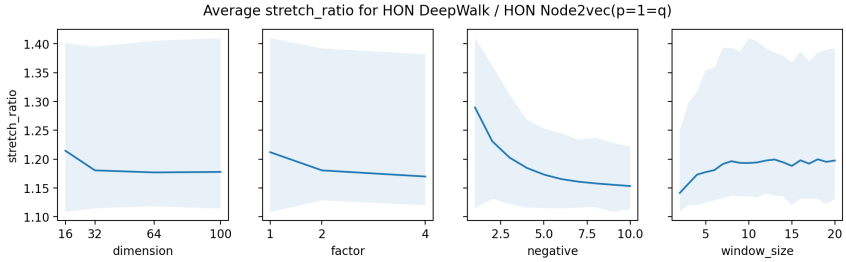
In [14]:

```
# Node2vec (7 hrs)
res_N2V = []
n2v_train_param = dict(num_iter=5, random_seed=100)
emb = None
for current in iter_cross_prod(p=[0.5,1,2,4], dimension=[16, 32, 64, 100], window_size=
range(1,21), factor=[1,2,4], negative=range(1,11)):
    if (emb is None) or (emb._dimension!=current['dimension']) or (emb._gen._p!=current
['p']):
        emb = HON_Node2vec_Embedding(gen, current['dimension'], p=current['p'], q=1/cur
rent['p'], reuse_walks=True)
    window_size = current['window_size']
    negative = current['negative']
    factor = current['factor']
    emb.train(num_walks=1000//window_size, walk_length=factor*window_size, window_size=
window_size, hs=False, negative=negative, **n2v_train_param)
    stretch = get_stretch_ratio(emb)
    config = dict(stretch_ratio=stretch, name='Node2vec', q=1/current['p'])
    config.update(current)
    res_N2V.append(config)
res_N2V = pd.DataFrame(res_N2V).sort_values('stretch_ratio', ascending=False)
disp_marginals(res_N2V, emb, title='Average stretch_ratio for HON Node2vec', filename=
'tmp/stretch_sim_N2V.png')
#res_N2V.head(10)
```



In [15]:

```
# DeepWalk -> see Node2vec(p=1=q)
res_DW = res_N2V[res_N2V['p']==1].drop(['p','q'], axis=1)
res_DW['name'] = 'DeepWalk'
disp_marginals(res_DW, title='Average stretch_ratio for HON DeepWalk / HON Node2vec(p=1
=q)', filename='tmp/stretch_sim_DW.png')
```



In [16]:

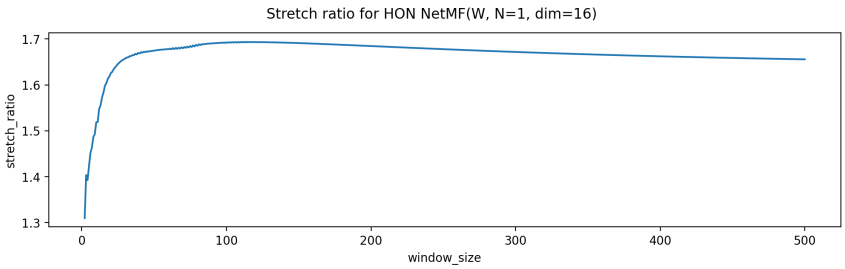
```
res_DW[res_DW['window_size']==2].head()
```

Out[16]:

	stretch_ratio	name	dimension	window_size	factor	negative
3640	1.248872	DeepWalk	64	2	2	1
2440	1.248437	DeepWalk	16	2	2	1
4240	1.248239	DeepWalk	100	2	2	1
3040	1.242888	DeepWalk	32	2	2	1
4241	1.188573	DeepWalk	100	2	2	2

In [17]:

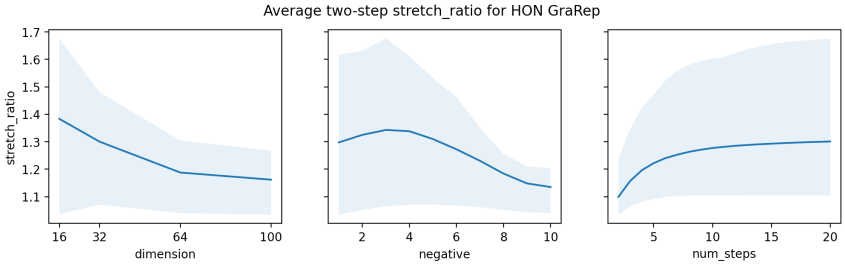
```
# NetMF (getting the optimal window size)
if True:
    res_NetMF2 = []
    emb = None
    # loop over dimension and window_size first to allow NetMF to reuse its PMI calcula
tion.
    for current in iter_cross_prod(dimension=[16], window_size=range(2,501), negative=[
1]):
        if (emb is None) or (emb._dimension!=current['dimension']):
            emb = HON_NetMF_Embedding(gen, dimension=current['dimension'], pairwise=Tru
e)
        emb.train(window_size=current['window_size'], negative=current['negative'])
        stretch = get_stretch_ratio(emb)
        config = dict(stretch_ratio=stretch, name='NetMF')
        config.update(current)
        res_NetMF2.append(config)
    res_NetMF2 = pd.DataFrame(res_NetMF2).sort_values('stretch_ratio', ascending=False)
    disp_stretch_W(res_NetMF2, emb, title='Stretch ratio for HON NetMF(W, N=1, dim=16)'
, filename='tmp/stretch_sim_NetMF_by_W.png')
    res_NetMF2.head(10)
```



GraRep

In [18]:

```
import warnings
# GraRep
res_GraRep = []
emb = None
# loop over dimension and window_size first to allow NetMF to reuse its PMI calculatio
n.
for current in iter_cross_prod(dimension=[16, 32, 64, 100], num_steps=range(1,21), nega
tive=range(1,11)):
    if (emb is None) or (emb._dimension!=current['dimension']) or (emb._num_steps!=curr
ent['num_steps']):
        emb = HON_GraRep_Embedding(gen, dimension=current['dimension'], num_steps=curre
nt['num_steps'], pairwise=True)
    with warnings.catch_warnings():
        warnings.simplefilter("ignore")
        emb.train(negative=current['negative'])
    stretch = get_stretch_ratio(emb, edge_distance=2)
    config = dict(stretch_ratio=stretch, name='GraRep')
    config.update(current)
    res_GraRep.append(config)
res_GraRep = pd.DataFrame(res_GraRep).sort_values('stretch_ratio', ascending=False)
disp_marginals(res_GraRep, emb, title='Average two-step stretch_ratio for HON GraRep',
filename='tmp/stretch_sim_GraRep.png')
#res_GraRep.head(10)
```



In [19]:

```
res_GraRep[res_GraRep['num_steps']==2].head()
```

Out[19]:

	stretch_ratio	name	dimension	num_steps	negative
13	1.237977	GraRep	16	2	4
14	1.235931	GraRep	16	2	5
12	1.235618	GraRep	16	2	3
15	1.228402	GraRep	16	2	6
11	1.222282	GraRep	16	2	2

C.8 SocioPatterns.ipynb

As discussed in § 4.2.2, we had to estimate the transition probabilities from the original data using `Python/data/models/SocioPatterns.ipynb`. Note that this requires pathpy2[2].

[2]We use version 2.2.0 installed from https://pypi.org/project/pathpy2/

Data preparation

SocioPatterns.org/Datasets (http://www.sociopatterns.org/datasets/)

In [ ]:

```
import os
base_dir = os.path.expanduser(r'~\Desktop\MScThesis\SocioPatterns')
work_contacts_dir = os.path.join(base_dir, r'Contacts in a workplace')
primary_school_dir = os.path.join(base_dir, r'Primary school temporal')
hospital_ward_dir = os.path.join(base_dir, r'Hospital ward')
out_dir = os.path.expanduser(r'~\Desktop\MScThesis\Python\data\models')
```

In [ ]:

```
# this workbook needs pathpy2 installed
%conda list pathpy
```

In [ ]:

```
import pathpy as pp
import numpy as np
import scipy.sparse.linalg as sla
from collections import defaultdict
```

In [ ]:

```
def iter_transitions(hon, include_subpaths:bool=True):
    "iterate over transition matrix of HigherOrderNetwork"
    mat = hon.transition_matrix(include_subpaths)
    if hon.order == 1:
        nodes = list((n,) for n in hon.nodes.keys())
    else:
        nodes = list(tuple(n.split(hon.separator)) for n in hon.nodes.keys())
    for r in range(mat.shape[0]):
        for ind in range(mat.indptr[r], mat.indptr[r+1]):
            yield nodes[mat.indices[ind]], nodes[r][-1], mat.data[ind] # yield (start,
 next_node, prob)
```

In [ ]:

```
# orig: Calc_q.ipynb
def calc_q(net, C, weighted=False, **kwargs):
    nodes = list(net.nodes.keys())
    mat = net.adjacency_matrix(weighted=weighted, **kwargs) # scipy.sparse.csc.csc_matr
ix
    mat_sum = 0
    mat_sum_by_row_C = defaultdict(float)
    mat_sum_by_col_C = defaultdict(float)
    q = 0
    for c in range(mat.shape[1]):
        c_C = C[nodes[c]] # class of current column's node
        for ind in range(mat.indptr[c], mat.indptr[c+1]):
            r = mat.indices[ind]
            v = mat.data[ind]
            # assert mat[r,c]==v
            r_C = C[nodes[r]] # class of current rows's node
            mat_sum += v
            mat_sum_by_row_C[r_C] += v
            mat_sum_by_col_C[c_C] += v
            if c_C == r_C:
                q += v
    q_exp = sum( v*mat_sum_by_col_C[c] for c,v in mat_sum_by_row_C.items() ) / (mat_sum
**2)
    q = q/mat_sum - q_exp
    q_max = 1 - q_exp
    return (q, q_max)
```

In [ ]:

```
def export_rules(filename, paths, config, metadata, max_order:int=1, replace_space=Fals
e, include_subpaths=True):
    "Export the transition probabilities (rules) up to the estimated order"
    mog = pp.MultiOrderModel(paths, max_order)
    estimated_order = mog.estimate_order(paths)
    print('Estimated order:', estimated_order)
    stats = dict()
    with open(filename,'w') as f:
        for order in range(1,estimated_order+1):
            hon = mog.layers[order]
            if order == 1:
                print('Exporting stationary distribution')
                probs = hon.transition_matrix(include_subpaths)
                _,ev = sla.eigs(probs, k=1, which='LM')
                ev = np.abs(ev).flatten()
                ev_sum = ev.sum()
                stat_dist = {n:v/ev_sum for n,v in zip(hon.nodes.keys(), ev)}
                for n,v in sorted(list(stat_dist.items())):
                    f.write('=> %s %r\n' % (n, v))
            print('Exporting rules for order',order)
            for start,next_node,prob in sorted(list(iter_transitions(hon, include_subpa
ths))):
                if replace_space:
                    next_node = next_node.replace(' ','_')
                    start = tuple( n.replace(' ','_') for n in start)
                for n in start:
                    f.write(n + ' ')
                f.write('=>')
                f.write(' %s %r\n' % (next_node,prob))
            # calc q, q_max
            node2cat = { n:metadata[n.split(',')[-1]] for n in hon.nodes } # last node
 is relevant for category
            q, q_max = calc_q(hon, node2cat, weighted=True, include_subpaths=include_su
bpaths)
            print(f'order {hon.order}: q={q}, q_max={q_max}, q/q_max={q/q_max}')
            stats.update({ f'q[{hon.order}]': q, f'q_max[{hon.order}]': q_max, f'q/q_ma
x[{hon.order}]': q/q_max })
    config_filename = os.path.splitext(filename)[0]+'.config'
    with open(config_filename,'w') as f:
        f.write('code\t%s\n' % os.path.join(os.getcwd(), 'SocioPatterns.ipynb'))
        for k,v in config.items():
            f.write('%s\t%s\n' % (k,v))
        delta_min,delta_sec = config['delta'] // 3, config['delta'] % 3 *20
        delta_time = f'{delta_min}:{delta_sec:02d}'
        delta_text = '%d min' % delta_min if delta_sec==0 else '%d sec' % delta_sec if
delta_min==0 else delta_time
        config_loc = dict(max_order=max_order, estimated_order=estimated_order, replace
_space=replace_space, include_subpaths=include_subpaths,
                          delta_time=delta_time, delta_text=delta_text)
        for k,v in config_loc.items():
            f.write('%s\t%s\n' % (k,v))
        for k,v in stats.items():
            f.write('%s\t%s\n' % (k,v))
```

Contacts in workplace

delta=90 finished in less than 6 hrs; delta=180 did not finish within 24 hrs.

In [ ]:

```
filename = os.path.join(work_contacts_dir, 'tij_InVS.dat')
c_workplace=dict(source=filename)
t_workplace = pp.TemporalNetwork()
with open(filename,'r') as f:
    for line in f:
        t,i,j = line.split()
        t_workplace.add_edge(i, j, int(t)//20)
        t_workplace.add_edge(j, i, int(t)//20)
print(t_workplace.summary())
```

In [ ]:

```
# copy metadata
m_workplace = dict()
with open(os.path.join(work_contacts_dir, 'metadata_InVS13.txt'),'r') as f:
    with open(os.path.join(out_dir, 'metadata_workplace.csv'),'w') as g:
        #g.write(f.read())
        for line in f:
            i,Ci = line.split()
            m_workplace[i]=Ci
            g.write(line)
```

In [ ]:

```
#extraction_param = dict(delta=90)
extraction_param = dict(delta=30)
c_workplace.update(extraction_param)
p_workplace = pp.path_extraction.paths_from_temporal_network_dag(t_workplace, **extract
ion_param)
```

In [ ]:

```
%time export_rules(os.path.join(out_dir, 'workplace_%d.csv' % extraction_param['delt
a']), p_workplace, c_workplace, m_workplace, max_order=4) # estimated_order=2
```

In [ ]:

```
del p_workplace
```

Primary school

Unable to extract paths for delta=2 (on a PC with 16GB RAM)

In [ ]:

```
filename = os.path.join(primary_school_dir, 'primaryschool.csv')
c_primaryschool = dict(source=filename)
t_primaryschool = pp.TemporalNetwork()
m_primaryschool = dict() # metadata
with open(filename,'r') as f:
    for line in f:
        t,i,j,Ci,Cj = line.split()
        t_primaryschool.add_edge(i, j, int(t)//20)
        t_primaryschool.add_edge(j, i, int(t)//20)
        m_primaryschool[i]=Ci
        m_primaryschool[j]=Cj
print(t_primaryschool.summary())
```

In [ ]:

```
# export metadata
with open(os.path.join(out_dir, 'metadata_primaryschool.csv'),'w') as g:
    for i,c in sorted(list(m_primaryschool.items())):
        g.write('%s\t%s\n' % (i,c))
```

In [ ]:

```
# delta=1 took 2 hours
# delta=2 uses > 50GB RAM
extraction_param = dict(delta=1)
c_primaryschool.update(extraction_param)
p_primaryschool = pp.path_extraction.paths_from_temporal_network_dag(t_primaryschool, *
*extraction_param)
```

In [ ]:

```
%time export_rules(os.path.join(out_dir, 'primaryschool_%d.csv') % extraction_param['de
lta'], p_primaryschool, c_primaryschool, m_primaryschool, max_order=3)
```

In [ ]:

```
del p_primaryschool
```

Hospital ward

delta=3 took 12 hrs to generate paths and 5 hrs to export rules

In [ ]:

```
filename = os.path.join(hospital_ward_dir, 'detailed_list_of_contacts_Hospital.dat')
c_hospitalward = dict(source=filename)
t_hospitalward = pp.TemporalNetwork()
m_hospitalward = dict()
with open(filename,'r') as f:
    for line in f:
        t,i,j,Si,Sj = line.split()
        t_hospitalward.add_edge(i, j, int(t)//20)
        t_hospitalward.add_edge(j, i, int(t)//20)
        m_hospitalward[i]=Si
        m_hospitalward[j]=Sj
print(t_hospitalward.summary())
```

In [ ]:

```
# export metadata
with open(os.path.join(out_dir, 'metadata_hospital.csv'),'w') as g:
    for i,c in sorted(list(m_hospitalward.items())):
        g.write('%s\t%s\n' % (i,c))
```

In [ ]:

```
#extraction_param = dict(delta=3)
extraction_param = dict(delta=2)
c_hospitalward.update(extraction_param)
p_hospitalward = pp.path_extraction.paths_from_temporal_network_dag(t_hospitalward, **e
xtraction_param)
```

In [ ]:

```
%time export_rules(os.path.join(out_dir, 'hospital_%d.csv' % extraction_param['delt
a']), p_hospitalward, c_hospitalward, m_hospitalward, max_order=4)
```

In [ ]:

```
del p_hospitalward
```

C.9 Classification_sim.ipynb

This jupyter notebook calculates the tables and plots for § 4.2. We display a shorter version, which includes two flags in the first code block to control the size of the output.

Init

In [1]:

```
comment = 'calculated in Classification_sim (short).ipynb'
verbose = False
tiny=True
```

In [2]:

```
%matplotlib inline
#%matplotlib notebook
import os
import pathpy as pp
import numpy as np
import pandas as pd
import math
from collections import defaultdict
import matplotlib.pyplot as plt
import seaborn as sns
```

In [3]:

```
from HigherOrderPathGenerator import HigherOrderPathGenerator, CrossValidation_HigherOr
derPathGenerator
from Embedding import ABCEmbedding, HON_DeepWalk_Embedding, HON_Node2vec_Embedding, HON
EM_Embedding, HON_NetMF_Embedding, HON_GraRep_Embedding, HON_Transition_Hierarchical_Em
bedding
#from SyntheticNetworks import create_lattice_2nd_order_dynamic
from Visualizations import Visualization, EmbeddingView, Lattice2D_EmbeddingView
from Datasets import init_generator
```

In [4]:

```
#filename='workplace_3.csv'
#filename='workplace_15.csv'
#filename='workplace_30.csv'
#filename='workplace_60.csv'
#filename='workplace_90.csv'
#
filename='primaryschool_1.csv'
#filename='hospital_1.csv'
#filename='hospital_2.csv'
#filename='hospital_3.csv'
#filename='temporal-clusters.csv'
#filename='shuffled-temporal-clusters.csv'
```

In [5]:

```
gen = init_generator(filename, verbose=verbose)
delta_text = gen._config['delta_text']
palette='Set1'
dim=64
metadata_name = list(gen._source_path_metadata.keys())[0]
metadata2_name = None
if filename=='primaryschool_1.csv':
    palette=sns.color_palette('Paired', n_colors = 11)
    palette[-1] = (0,0,0) # replace yellow by black
    metadata_name='Class'
    metadata2_name = 'Role'
elif filename=='temporal-clusters.csv' or filename=='shuffled-temporal-clusters.csv':
    dim=16
    palette = sns.color_palette(['#0000FF','#00CC00','#FF0000'])
filename_noext = f'tmp/{os.path.splitext(filename)[0]}-dim{dim}'
```

```
97223 rules read
[]
[]
```

In [6]:

```
gen_FON = gen.to_FON()
```

In [7]:

```
if not os.path.exists('tmp'):
    os.makedirs('tmp')
```

Embeddings

In [8]:

```
embeddings_list = []
```

In [9]:

```
emb_N10 = HON_NetMF_Embedding(gen, dim, pairwise=True)
%time emb_N10.train(window_size=10, negative=1, optimized=True)
emb_N10._id = 'HON NetMF(W=10)'
embeddings_list.append(emb_N10)
```

```
Wall time: 8min 8s
```

In [10]:

```
emb_N10_FON = HON_NetMF_Embedding(gen_FON, dim, pairwise=True)
%time emb_N10_FON.train(window_size=10, negative=1, optimized=True)
emb_N10_FON._id = 'FON NetMF(W=10)'
embeddings_list.append(emb_N10_FON)
```

```
Wall time: 1min 24s
```

In [11]:

```
emb_N2 = HON_NetMF_Embedding(gen, dim, pairwise=True)
%time emb_N2.train(window_size=2, negative=1, optimized=True)
emb_N2._id = 'HON NetMF(W=2)'
embeddings_list.append(emb_N2)
```

```
Wall time: 1.25 s
```

In [12]:

```
emb_N2_FON = HON_NetMF_Embedding(gen_FON, dim, pairwise=True)
%time emb_N2_FON.train(window_size=2, negative=1, optimized=True)
emb_N2_FON._id = 'FON NetMF(W=2)'
embeddings_list.append(emb_N2_FON)
```

```
Wall time: 3.64 s
```

In [13]:

```
emb_G4 = HON_GraRep_Embedding(gen, dim//4, num_steps=4, pairwise=True, neg_stationary=F
alse)
%time emb_G4.train(negative=1, normalize=False) # LogisticRegression performs poorly if
normalize=True
emb_G4._id = 'HON GraRep(S=4)'
embeddings_list.append(emb_G4)
```

```
Wall time: 1min 17s
```

In [14]:

```
emb_G4_FON = HON_GraRep_Embedding(gen_FON, dim//4, num_steps=4, pairwise=True, neg_stat
ionary=False)
%time emb_G4_FON.train(negative=1, normalize=False) # LogisticRegression performs poorl
y if normalize=True
emb_G4_FON._id = 'FON GraRep(S=4)'
embeddings_list.append(emb_G4_FON)
```

```
Wall time: 23.3 s
```

In [15]:

```
deepwalk_para = dict(num_walks=100, hs=False, negative=1, random_seed=1)
emb_D10 = HON_DeepWalk_Embedding(gen, dim)
%time emb_D10.train(window_size=10, **deepwalk_para)
emb_D10._id = 'HON DeepWalk'
embeddings_list.append(emb_D10)
```

```
Wall time: 32.1 s
```

In [16]:

```
emb_D10_FON = HON_DeepWalk_Embedding(gen_FON, dim)
%time emb_D10_FON.train(window_size=10, **deepwalk_para)
emb_D10_FON._id = 'FON DeepWalk'
embeddings_list.append(emb_D10_FON)
```

```
Wall time: 29.3 s
```

In [17]:

```
node2vec_para = dict(num_walks=100, hs=False, negative=1, random_seed=1)
emb_N2V_10a = HON_Node2vec_Embedding(gen, dim, p=0.5, q=2)
%time emb_N2V_10a.train(window_size=10, **node2vec_para)
emb_N2V_10a._id = 'HON Node2vec(p=0.5)'
embeddings_list.append(emb_N2V_10a)
```

```
Wall time: 32.1 s
```

In [18]:

```
emb_N2V_10a_FON = HON_Node2vec_Embedding(gen_FON, dim, p=0.5, q=2)
%time emb_N2V_10a_FON.train(window_size=10, **node2vec_para)
emb_N2V_10a_FON._id = 'FON Node2vec(p=0.5)'
embeddings_list.append(emb_N2V_10a_FON)
```

```
Wall time: 32.8 s
```

In [19]:

```
emb_N2V_10b = HON_Node2vec_Embedding(gen, dim, p=2, q=0.5)
%time emb_N2V_10b.train(window_size=10, **node2vec_para)
emb_N2V_10b._id = 'HON Node2vec(p=2)'
embeddings_list.append(emb_N2V_10b)
```

```
Wall time: 33.6 s
```

In [20]:

```
emb_N2V_10b_FON = HON_Node2vec_Embedding(gen_FON, dim, p=2, q=0.5)
%time emb_N2V_10b_FON.train(window_size=10, **node2vec_para)
emb_N2V_10b_FON._id = 'FON Node2vec(p=2)'
embeddings_list.append(emb_N2V_10b_FON)
```

```
Wall time: 33.6 s
```

In [21]:

```
emb_H = HONEM_Embedding(gen, dim)
%time emb_H.train()
emb_H._id = 'HONEM'
embeddings_list.append(emb_H)
```

```
Wall time: 358 ms
```

In [22]:

```
embeddings = { e._id:e for e in embeddings_list}
```

Visualization

In [23]:

```
embedding_views = [ EmbeddingView(emb, use_source=True) for emb in embeddings_list ]
```

In [24]:

```
def plot_ev(ev, filename=None, random_state=1, comment=comment):
    vis = ev.visualize_TSNE(random_state=random_state, title=ev._emb._id + ', TSNE')
    metadata_classes = sorted(list(ev[metadata_name].unique()))
    plot_args = dict(hue=metadata_name, palette=palette, hue_order=metadata_classes)
    if metadata2_name is not None:
        plot_args['style'] = metadata2_name
    if tiny:
        plot_args['figsize']=(4.5,3)
        plot_args['dpi']=75
    vis.plot1(**plot_args)
    if filename is not None:
        vis.save_describe(filename, comment)
if verbose:
    for ev in embedding_views:
        plot_ev(ev, random_state=1, filename=None)
```

Classification

In [25]:

```
from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
```

In [26]:

```
models = {
    'LR': LogisticRegression(random_state=1),
    'SVC': SVC(random_state=1, probability=True),
    'RF': RandomForestClassifier(random_state=1)
}
```

In [27]:

```
from sklearn.model_selection import KFold
from sklearn.metrics import f1_score
def get_stats(ev, model, n_splits=5, n_repeat=10, verbose=True, target_name=metadata_na
me):
    stats = defaultdict(list)
    X = ev.embedding.values
    y = ev[target_name].values
    #y_classes = sorted(np.unique(x))
    for random_state in range(n_repeat):
        if verbose: print('.', end='')
        y_true=[]
        y_pred=[]
        scores=[]
        kf = KFold(n_splits, shuffle=True, random_state=random_state)
        for train_index, test_index in kf.split(X):
            X_train, X_test = X[train_index], X[test_index]
            y_train, y_test = y[train_index], y[test_index]
            model.fit(X_train, y_train)
            scores.append(model.score(X_test, y_test))
            y_true.extend(y_test)
            y_pred.extend(model.predict(X_test))
        stats['score'].append(np.mean(scores))
        stats['f1-micro'].append(f1_score(y_true, y_pred, average='micro'))
        stats['f1-macro'].append(f1_score(y_true, y_pred, average='macro'))
    if verbose: print()
    res = { k: np.mean(v) for k,v in stats.items() }
    res.update({k+' (min)': np.min(v) for k,v in stats.items() })
    res.update({k+' (max)': np.max(v) for k,v in stats.items() })
    res.update({k+' (std)': np.std(v) for k,v in stats.items() })
    return res
```

In [28]:

```
stats_list = []
for ev in embedding_views:
    if verbose: print('Embedding', ev._emb._id)
    for model_name, model in models.items():
        if verbose: print('- Model', model_name, end='')
        stats = dict(embedding=ev._emb._id, model=model_name)
        stats.update(get_stats(ev, model, verbose=verbose))
        stats_list.append(stats)
stats_df = pd.DataFrame(stats_list).set_index(['model','embedding'])
if verbose:
    stats_df
```

In [29]:

```
stats4tex = stats_df[['f1-macro','f1-micro']].reset_index().set_index(['embedding','mod
el']).unstack()
if verbose:
    stats4tex
```

In [30]:

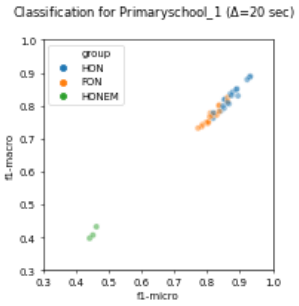
```
def summarize(col='f1-macro'):
    df = stats_df[[col]].sort_values(col, ascending=False).copy().reset_index(drop=Fals
e)
    df['group'] = df['embedding'].map(lambda x:x.split(' ')[0])
    df['rank'] = df.index+1
    df['score'] = col
    df['value'] = df[col]
    df['dataset'] = gen._id
    return df.groupby(['dataset','score','group'])[['value', 'rank']].mean()
summarize().append(summarize('f1-micro'))
```

Out[30]:

			value	rank
dataset	score	group		
primaryschool_1	f1-macro	FON	0.769721	25.611111
		HON	0.825248	11.388889
		HONEM	0.412202	38.000000
	f1-micro	FON	0.815680	26.166667
		HON	0.871786	10.833333
		HONEM	0.452342	38.000000

In [31]:

```
with open(filename_noext + '.tex', 'w') as f:
    gen.write_config(f, comment=comment, prefix='% ', sep='=')
    f.write('%\n% Embeddings:\n')
    for emb in embeddings.values():
        f.write(f'% {emb._id}={repr(emb.config)}\n')
    f.write('\n')
    stats4tex.to_latex(f, float_format="%.3f", index_names=False)
# export summary
summarize('f1-macro').append(summarize('f1-micro')).to_csv(filename_noext + '.csv', sep
='\t', encoding='utf-16')
# export plot
df = stats_df.copy().reset_index(drop=False)
df['group'] = df['embedding'].map(lambda x:x.split(' ')[0])
fig = plt.figure(figsize=(4, 4.3),dpi=50 if tiny else 400)
fig.suptitle('Classification for ' + gen._id.title() + ' ($\Delta$=' + delta_text + ')'
)
ax = sns.scatterplot(x='f1-micro', y='f1-macro', hue='group', data=df, ax=fig.gca(), al
pha=0.5)
f1_min = min(df['f1-micro'].min(), df['f1-macro'].min())
f1_max = max(df['f1-micro'].max(), df['f1-macro'].max())
f1_min = np.floor(f1_min*10)/10
f1_max = np.ceil(f1_max*10)/10
ax.set_xlim(f1_min, f1_max)
ax.set_ylim(f1_min, f1_max)
ax.set_aspect(1)
ax.figure.savefig(filename_noext + '.png')
```



In [32]:

```
sorted_f1_micro = stats_df['f1-micro'].sort_values(ascending=False)
sorted_f1_micro.to_csv(filename_noext + '_f1-micro.csv', sep='\t', encoding='utf-16')
if verbose:
    sorted_f1_micro
```

In [33]:

```
sorted_f1_macro = stats_df['f1-macro'].sort_values(ascending=False)
sorted_f1_macro.to_csv(filename_noext + '_f1-macro.csv', sep='\t', encoding='utf-16')
if verbose:
    sorted_f1_macro
```

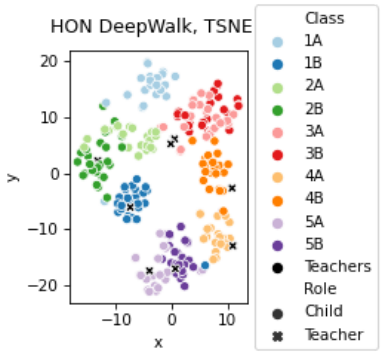
In [34]:

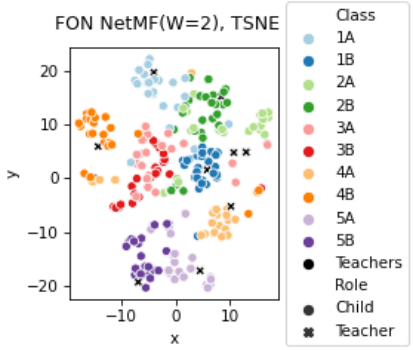
```
sorted_f1 = (stats_df['f1-micro'] + stats_df['f1-macro']).sort_values(ascending=False)
best_HON =sorted_f1[sorted_f1.index.map(lambda x:x[1].startswith('HON '))].index[0]
best_FON =sorted_f1[sorted_f1.index.map(lambda x:x[1].startswith('FON '))].index[0]
print('best HON model and embedding', best_HON)
print('best FON model and embedding', best_FON)
```

```
best HON model and embedding ('LR', 'HON DeepWalk')
best FON model and embedding ('LR', 'FON NetMF(W=2)')
```

In [35]:

```
# export tsne visualization of best embeddings
plot_ev(EmbeddingView(embeddings[best_HON[1]], use_source=True), random_state=2, filena
me=f'{filename_noext}_tsne_hon.png')
plot_ev(EmbeddingView(embeddings[best_FON[1]], use_source=True), random_state=1, filena
me=f'{filename_noext}_tsne_fon.png')
```





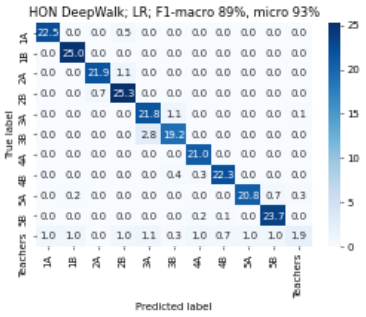
Confusion matrix

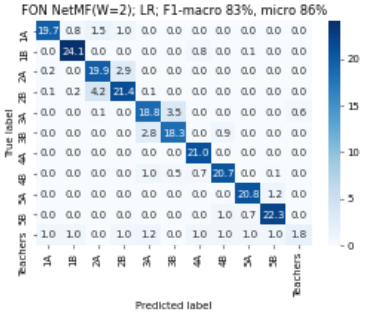
In [36]:

```
from sklearn.metrics import confusion_matrix
from sklearn.base import BaseEstimator
def create_confusion_matrix(ev: EmbeddingView, model: BaseEstimator, n_splits:int=5, n_
repeat=10, plot:bool=False, title=None, target_name=metadata_name):
    if verbose:
        print(ev._emb._id, model)
    X = ev.embedding.values
    y = ev[target_name].values
    classes = sorted(np.unique(y))
    y_true, y_pred=[], []
    for random_state in range(n_repeat):
        kf = KFold(n_splits, shuffle=True, random_state=random_state)
        for train_index, test_index in kf.split(X):
            X_train, X_test = X[train_index], X[test_index]
            y_train, y_test = y[train_index], y[test_index]
            model.fit(X_train, y_train)
            y_true.extend(y_test)
            y_pred.extend(model.predict(X_test))
    cm = confusion_matrix(y_true, y_pred, labels=classes) / n_repeat
    if plot:
        plt.figure(figsize=(6,4) if tiny else(7,5), dpi=50 if tiny else 100)
        if title is not None:
            plt.title(title)
        sns.heatmap(cm, annot=True, fmt='.1f', cmap=plt.cm.Blues, xticklabels=classes,
yticklabels=classes)
        plt.xlabel('Predicted label')
        plt.ylabel('True label')
    return pd.DataFrame(cm, index=pd.MultiIndex.from_product([['True label'], classes
]),
                        columns=pd.MultiIndex.from_product([['Predicted label'], classe
s]))
```

In [37]:

```
def export_confusion_matrix(best, filename=None, comment=comment):
    title = f'{best[1]}; {best[0]}; F1-macro {sorted_f1_macro[best]:.0%}, micro {sorted
_f1_micro[best]:.0%}'
    emb = embeddings[best[1]]
    cf_para = dict(model=models[best[0]], n_splits=5, n_repeat=10)
    data = create_confusion_matrix(EmbeddingView(emb, use_source=True), plot=True, titl
e=title, **cf_para)
    if filename is not None:
        plt.gcf().savefig(filename, bbox_inches='tight')
        with open(filename + '.txt','w') as f:
            comment_loc = (comment + ('' if comment=='' else '\n\n')
                + f'F1-macro\t{sorted_f1_macro[best]:f}\nF1-micro\t{sorted_f1_micro[bes
t]:f}')
            emb.write_config(f, comment=comment_loc, prefix='', sep='\t')
            f.write('\nConfusion matrix:\n')
            for k,v in cf_para.items():
                f.write('%s\t%s\n' % (k,v))
            f.write('\n\nData:\n')
            f.write(data.to_csv(line_terminator='\n'))
export_confusion_matrix(best_HON, filename=f'{filename_noext}_confusion_hon.png')
export_confusion_matrix(best_FON, filename=f'{filename_noext}_confusion_fon.png')
```





AUC / ROC

In [38]:

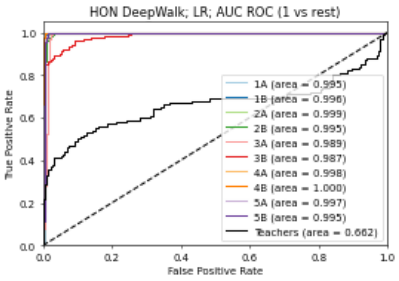
```
# https://stackoverflow.com/questions/45332410/sklearn-roc-for-multiclass-classificatio
n
from sklearn.metrics import roc_curve, auc
def create_auc_roc_old(ev: EmbeddingView, model: BaseEstimator, n_splits:int=5, n_repea
t=10, title=None, plot:bool=False, target_name=metadata_name):
    X = ev.embedding.values
    #y = ev.target.values
    y = ev[target_name].values
    y_true, y_score = [], []
    for random_state in range(n_repeat):
        kf = KFold(n_splits, shuffle=True, random_state=random_state)
        for train_index, test_index in kf.split(X):
            X_train, X_test = X[train_index], X[test_index]
            y_train, y_test = y[train_index], y[test_index]
            model.fit(X_train, y_train)
            y_true.extend(y_test)
            y_score.append(model.decision_function(X_test)) # corresponding labels are
 in model.classes_
    y_score = np.concatenate(y_score, axis=0) # rbind(...)
    y_true = np.array(y_true)

    fpr, tpr, roc_auc = dict(), dict(), dict()
    for i,c in enumerate(model.classes_):
        # roc_curve for y_true == c vs. rest
        fpr_, tpr_, _ = roc_curve(y_true == c, y_score[:,i])
        fpr[c] = fpr_
        tpr[c] = tpr_
        roc_auc[c] = auc(fpr_, tpr_)
    if plot:
        colors = sns.color_palette(palette, len(model.classes_))
        plt.figure(dpi=200)
        for c in model.classes_: # or sorted(np.unique(y))
            plt.plot(fpr[c], tpr[c], label='%s (area = %0.2f)' % (c,roc_auc[c]), color=
colors.pop(0))
        plt.plot([0, 1], [0, 1], 'k--')
        plt.xlim([0.0, 1.0])
        plt.ylim([0.0, 1.05])
        plt.xlabel('False Positive Rate')
        plt.ylabel('True Positive Rate')
        if title is None:
            plt.title('Receiver operating characteristic (1 vs rest) - %d times %d-fold
CV' % (n_repeat, n_splits))
        else:
            plt.title(title)
        plt.legend(loc="lower right")
        #plt.show()

    return pd.Series(roc_auc, index=model.classes_, name='auc')#, fpr, tpr
```

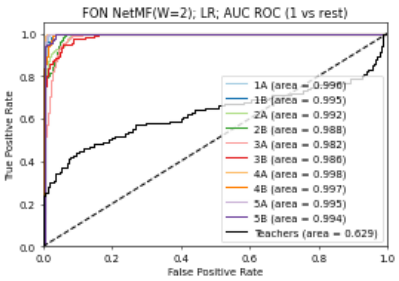
In [39]:

```
from sklearn.metrics import roc_curve, auc, roc_auc_score
def create_auc_roc(ev: EmbeddingView, model: BaseEstimator, n_splits:int=5, n_repeat=10
, title=None, plot:bool=False, target_name=metadata_name, check=True):
    X = ev.embedding.values
    #y = ev.target.values
    y = ev[target_name].values
    y_true, y_prob = [], []
    for random_state in range(n_repeat):
        kf = KFold(n_splits, shuffle=True, random_state=random_state)
        for train_index, test_index in kf.split(X):
            X_train, X_test = X[train_index], X[test_index]
            y_train, y_test = y[train_index], y[test_index]
            model.fit(X_train, y_train)
            y_true.extend(y_test)
            y_prob.append(model.predict_proba(X_test)) # corresponding labels are in mo
del.classes_
    y_prob = np.concatenate(y_prob, axis=0) # rbind(...)
    y_true = np.array(y_true)

    fpr, tpr, roc_auc = dict(), dict(), dict()
    for i,c in enumerate(model.classes_):
        # roc_curve for y_true == c vs. rest
        fpr_, tpr_, _ = roc_curve(y_true == c, y_prob[:,i])
        fpr[c] = fpr_
        tpr[c] = tpr_
        roc_auc[c] = auc(fpr_, tpr_)
    if check:
        # "Area under ROC for the multiclass problem" acording to sklearn documentation
        # https://scikit-learn.org/stable/auto_examples/model_selection/plot_roc.html
        macro_roc_auc_ovr = roc_auc_score(y_true, y_prob, multi_class='ovr', average='m
acro')
        # getting the 'macro'-average of roc_auc, which is supposed to calculate the sa
me
        auc_mean = np.mean(list(roc_auc.values()))
        if abs(auc_mean - macro_roc_auc_ovr) > 1e-16:
            print('The indicidual AUCs are inconsistent with roc_auc_score(ovr,macro)=%
f'%macro_roc_auc_ovr)
    if plot:
        colors = sns.color_palette(palette, len(model.classes_))
        plt.figure(dpi=50 if tiny else 200)
        for c in model.classes_: # or sorted(np.unique(y))
            plt.plot(fpr[c], tpr[c], label='%s (area = %0.3f)' % (c,roc_auc[c]), color=
colors.pop(0))
        plt.plot([0, 1], [0, 1], 'k--')
        plt.xlim([0.0, 1.0])
        plt.ylim([0.0, 1.05])
        plt.xlabel('False Positive Rate')
        plt.ylabel('True Positive Rate')
        if title is None:
            plt.title('Receiver operating characteristic (1 vs rest) - %d times %d-fold
CV' % (n_repeat, n_splits))
        else:
            plt.title(title)
        plt.legend(loc="lower right")
        #plt.show()

    return pd.Series(roc_auc, index=model.classes_, name='auc')#, fpr, tpr
```

In [40]:

```
def export_auc_roc(best, filename=None, comment=comment):
    title = f'{best[1]}; {best[0]}; AUC ROC (1 vs rest)'
    emb = embeddings[best[1]]
    auc_para = dict(model=models[best[0]], n_splits=5, n_repeat=10)
    data = create_auc_roc(EmbeddingView(emb, use_source=True), title=title, plot=True,
**auc_para)
    if filename is not None:
        plt.gcf().savefig(filename, bbox_inches='tight')
        with open(filename + '.txt','w') as f:
            comment_loc = (comment + ('' if comment=='' else '\n\n')
                + f'F1-macro\t{sorted_f1_macro[best]:f}\nF1-micro\t{sorted_f1_micro[bes
t]:f}\n'
                + f'macro_roc_auc_ovr\t{data.mean():f}')
            emb.write_config(f, comment=comment_loc, prefix='', sep='\t')
            f.write('\nAUC ROC:\n')
            for k,v in auc_para.items():
                f.write('%s\t%s\n' % (k,v))
            f.write('\n\nData:\n')
            f.write(data.to_csv(line_terminator='\n'))
export_auc_roc(best_HON, filename=f'{filename_noext}_auc-roc_hon.png')
export_auc_roc(best_FON, filename=f'{filename_noext}_auc-roc_fon.png')
```





Entropy

In [41]:

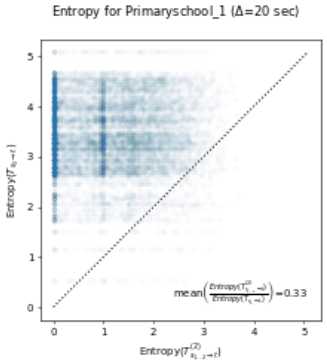
```
entropies = dict()
for key in gen.rule_keys:
    tmp = 0
    for _,_,prob in gen.transition_probs(start=key):
        tmp += prob * math.log2(prob)
    entropies[key] = -tmp
```

In [42]:

```
entropy_data = [
    dict(start=str(start), entropy=entropy,
         parent_entropy=entropies[start[:-1]],
         start_len=len(start)
        )
    for start,entropy in entropies.items() if len(start)>0
]
entropy_data = pd.DataFrame(entropy_data).set_index('start')
entropy_data['entropy_ratio'] = entropy_data['entropy']/entropy_data['parent_entropy']
```

In [43]:

```
entropy_data2 = entropy_data[entropy_data['start_len']==2]
avg_ratio_desc = 'mean$\\left(\\frac{Entropy(T^{(2)}_{s_{1..2}\\rightarrow t})}{Entropy
(T_{s_2\\rightarrow t})}\\right)$=%.2f' % entropy_data["entropy_ratio"].mean()
fig = plt.figure(figsize=(5,5), dpi=50 if tiny else 400)
alpha = 0.01 if filename == 'primaryschool_1.csv' else 0.1
ax = entropy_data2.plot.scatter('entropy','parent_entropy', alpha=alpha, ax=fig.gca())
max_entropy = entropy_data2['parent_entropy'].max()
ax.set_aspect(1)
ax.set_xlabel('Entropy($T^{(2)}_{s_{1..2}\\rightarrow t}$)')
ax.set_ylabel('Entropy($T_{s_2\\rightarrow t}$)')
ax.plot([0, max_entropy], [0, max_entropy], 'k:')
ax.figure.suptitle('Entropy for ' + gen._id.title() + ' ($\\Delta$=' + delta_text + ')'
)
ax.annotate(avg_ratio_desc, (max_entropy,0), ha='right', va='bottom')
fig.savefig(f'tmp/{os.path.splitext(filename)[0]}_entropy.png')
```



In [44]:

```
entropy_data['entropy'].mean()
```

Out[44]:

```
1.1137836903958092
```

In [45]:

```
entropy_data['parent_entropy'].mean()
```

Out[45]:

```
3.501841638726696
```

C.10 Plot_ExpClass.ipynb

This jupyter notebook calculates figure 4.11.

In [1]:

```
comment = 'calculated in Plots_ExpClass.ipynb'
```

In [2]:

```
%matplotlib inline
#%matplotlib notebook
import os
import pandas as pd
from collections import defaultdict
import matplotlib.pyplot as plt
import seaborn as sns
```

In [3]:

```
from Datasets import init_generator # returns HigherOrderPathGenerator
```

In [4]:

```
gen_school = init_generator('primaryschool_1.csv')
```

```
97223 rules read
[]
[]
```

In [5]:

```
if not os.path.exists('tmp'):
    os.makedirs('tmp')
```

Check child-teacher-child interactions

In [6]:

```
def is_teacher(node):
    return gen_school._target_node_metadata['Role'][node] == 'Teacher'
def is_child(node):
    return gen_school._target_node_metadata['Role'][node] == 'Child'
def get_class(node):
    return gen_school._target_node_metadata['Class'][node]
```

In [7]:

```
###ctc_paths = list() # child-teacher-child paths
def get_class_class_probs_via_teacher(HON=True, title=None, filename=None, transition_p
robs=False, figsize=(7,5), dpi=200, bbox_inches='tight', **kwargs):
    """
    returns a dictionary { (c1,c2) : prob}
    where (c1,c2) is a pair of classes
    and prob = P(class(n1) = c1, class(n3)=c2 | is_child(n1), is_teacher(n2), is_child
(n3))
    where (n1,n2,n3) is a random path of length 3 starting with n1 ~ stationary
    """
    gen = gen_school if HON else gen_school.to_FON()
    total_prob = 0
    cp = defaultdict(float) # prob of class
    ccp = defaultdict(float) # prob of pair of classes
    for (n1,n2,n3),prob in gen.path_probs(start=(), num_steps=3):
        if is_child(n1) and is_teacher(n2) and is_child(n3):
            c1 = get_class(n1)
            c3 = get_class(n3)
            total_prob += prob
            cp[c1] += prob
            ccp[(c1,c3)] += prob
    print('total probability', total_prob)
    if transition_probs:
        ccp = { cc:prob/cp[cc[0]] for cc,prob in ccp.items()} # transition probability
 c1->c2
    else:
        ccp = { cc:prob/total_prob for cc,prob in ccp.items()} # probability of pairs
    res = pd.Series(ccp).unstack()
    if filename is not None:
        fig = plt.figure(figsize=figsize, dpi=dpi)
        if title is not None:
            fig.suptitle(title)
        classes = sorted(cp.keys())
        ax = fig.gca()
        sns.heatmap(res, annot=True, fmt='.2f', cmap=plt.cm.Blues, xticklabels=classes,
yticklabels=classes, ax=ax)
        if transition_probs:
            ax.set_xlabel('target')
            ax.set_ylabel('source')
        else:
            ax.set_xlabel('class of $child_2$')
            ax.set_ylabel('class of $child_1$')
        fig.savefig(filename, bbox_inches=bbox_inches, **kwargs)
        with open(filename + '.txt','w') as f:
            gen.write_config(f, comment=comment, prefix='', sep='\t')
            f.write('\n\nProbability of a child->teacher->child path %f\n' % total_prob
)
            if transition_probs:
                f.write('\nTransition probabilities:\n')
            else:
                f.write('\nPair probabilities:\n')
            f.write(res.to_csv(line_terminator='\n'))
    return res
```

In [ ]:

```
title_HON = 'Cond. probability of Child-Teacher-Child interaction by class (HON)'
c2c_HON = get_class_class_probs_via_teacher(True, title_HON, filename='tmp/school_hon_i
nteraction_via_teacher.png')
```

In [ ]:

```
title_FON = 'Cond. probability of Child-Teacher-Child interaction by class (FON)'
c2c_FON = get_class_class_probs_via_teacher(False, title_FON, filename='tmp/school_fon_
interaction_via_teacher.png')
```

Get frequencies by class from stationary distribution

In [10]:

```
class_probs = defaultdict(float)
for _,n,prob in gen_school.transition_probs(start=()): # stationary distribution
    class_probs[get_class(n)] += prob
class_probs
```

Out[10]:

```
defaultdict(float,
            {'5B': 0.08779785547204058,
             '5A': 0.04456183510387105,
             '4A': 0.04474650478145715,
             'Teachers': 0.004609362345706892,
             '3B': 0.15727510236331413,
             '4B': 0.05062492826777912,
             '2A': 0.11235273025299394,
             '1B': 0.2166310296376714,
             '2B': 0.12849865639303715,
             '1A': 0.04554663582826103,
             '3A': 0.10735535955386742})
```

C.11 ProbabilityPrediction.ipynb

This jupyter notebook calculates the simulation and plots for § B.1.

Init

In [1]:
```
verbose=False
```

In [2]:
```
%matplotlib inline
#%matplotlib notebook
import os
import pathpy as pp
import numpy as np
import pandas as pd
import math
from collections import defaultdict
import matplotlib.pyplot as plt
#from tqdm import tqdm
from tqdm.notebook import tqdm
```

In [3]:
```
from HigherOrderPathGenerator import HigherOrderPathGenerator, CrossValidation
_HigherOrderPathGenerator, ABCHigherOrderPathGenerator
from Embedding import ABCEmbedding, HON_DeepWalk_Embedding, HONEM_Embedding, H
ON_NetMF_Embedding, HON_GraRep_Embedding, HON_Transition_Hierarchical_Embeddin
g
from Visualizations import Visualization, EmbeddingView, Lattice2D_EmbeddingVi
ew
from Datasets import init_generator
```

In [4]:
```
gen_HON = init_generator('primaryschool_1.csv')
#gen_HON = init_generator('workplace_30.csv')
#gen_HON = init_generator('hospital_1.csv')
```

```
97223 rules read
[]
[]
```

In [5]:
```
gen_FON = gen_HON.to_FON(gen_HON._id + ' (FON)')
```

Embeddings

In [6]:
```
class embedding_builder(object):
    "Instantiates an embedding and trains it."
    def __init__(self, name:str, init_para=None, **train_para):
        self._name = name
        self._init_para = dict() if init_para is None else init_para
        self._train_para = train_para

    def build(self, gen: ABCHigherOrderPathGenerator, dimension:int = 128):
        if self._name == 'NetMF':
            emb = HON_NetMF_Embedding(gen, dimension, **self._init_para)
            emb.train(**self._train_para)
        elif self._name == 'GraRep':
            emb = HON_GraRep_Embedding(gen, dimension, **self._init_para)
            emb.train(**self._train_para)
        elif self._name == 'Experiment':
            #emb = HON_Transition_Hierarchical_Embedding(gen, dimension, **sel
f._init_para)
            emb = HON_CV_Transition_Hierarchical_Embedding(gen, dimension, **s
elf._init_para)
            emb.train(**self._train_para)
        elif self._name == 'DeepWalk':
            emb = HON_DeepWalk_Embedding(gen, dimension)
        elif self._name == 'HONEM':
            emb = HONEM_Embedding(gen, dimension)
            emb.train()
        else:
            assert False, 'Invalid name %s' % name
        return emb

    @staticmethod
    def get_true_probs(gen: ABCHigherOrderPathGenerator, excluded_edges):
        res = defaultdict(dict) # use nested dictionaries, because decode is e
xpensive
        for source in gen.source_paths_len1:
            for _, next_node, prob in gen.transition_probs(source):
                if (source[-1],next_node) in excluded_edges:
                    res[source][next_node]=prob
        return res

    def evaluate_pairs(self, true_probs, gen_build: ABCHigherOrderPathGenerato
r, dimension: int = 128, **kwargs):
        emb = self.build(gen_build, dimension)
        node2str = emb.key2str if emb._symmetric else emb.node2str
        source2str = emb.key2str if emb._symmetric else emb.path2str
        res = []
        for source, probs in true_probs.items():
            source_str = source2str(source)
            predicted_probs = emb.decode_path(source, **kwargs)
            for next_node, prob in probs.items():
                next_node_str = node2str(next_node)
                res.append(dict(source=source_str, target=next_node_str, true_
prob=prob, pred_prob=predicted_probs[next_node_str]))
        return res

    def evaluate(self, true_probs, gen_build: ABCHigherOrderPathGenerator, dim
ension:int = 128, **kwargs):
```

```
        emb = self.build(gen_build, dimension)
        node2str = emb.key2str if emb._symmetric else emb.node2str
        SSE = 0
        SSElog = 0
        min_prob=1e-10
        for source, probs in true_probs.items():
            predicted_probs = emb.decode_path(source, **kwargs)
            for next_node, prob in probs.items():
                SSE += (prob - predicted_probs[node2str(next_node)])**2
                SSElog += (math.log2(max(prob,min_prob)) - math.log2(max(predi
cted_probs[node2str(next_node)],min_prob)))**2
        return dict(SSE=SSE,SSElog=SSElog) # include time? include SSE of log
 probabilities?
```

In [7]:
```
class HON_CV_Transition_Hierarchical_Embedding(HON_Transition_Hierarchical_Emb
edding):
    """In a cross-validation setting, some edges should be excluded from train
ing the embedding.
    While this is impossible for methods based on (unweighted) matrix factoriz
ation using SVD,
    this is simple for methods based on weighted matrix factorization using SG
D.

    The trick is to skip the update step if it happens to coincide with one of
the excluded edges.
    """
    def __init__(self, gen, dimension: int=128, node_hierarchy='calc', seed=No
ne, neg_stationary:bool=True):
        super().__init__(gen, dimension, node_hierarchy, seed, neg_stationary)
        if type(gen) == CrossValidation_HigherOrderPathGenerator:
            self._excluded_edges = { (iu,iv) for u,iu in self._source_paths.it
ems() for v,iv in self._target_nodes.items()
                      if (u[-1],v) in gen._excluded_edges }
        else:
            self._excluded_edges = set() # warn?

    def _update(self, iu, iv, label, learning_rate):
        if (iu,iv) in self._excluded_edges:
            return # skip training (both positive and negative samples) for th
e excluded edges
        super()._update(iu, iv, label, learning_rate)
```

In [8]:
```
dimensions = [16, 32, 64, 128]
```

In [9]:
```
decode_args = dict(use_neighborhood=True, no_self_loops=True, normalize=True,
step=1) # decode ignores additional parameters
```

In [10]:
```
netmf_pairs = dict(pairwise=True)
grarep_para = dict(num_steps=2, pairwise=True)
#deepwalk10_para = dict(num_walks=100, walk_length=100, window_size=10, negati
ve=5)
embedding_builders = {
    'NetMF(W1,N1)': embedding_builder('NetMF', init_para=netmf_pairs, window_s
ize=1, negative=1),
    'NetMF(W1,N5)': embedding_builder('NetMF', init_para=netmf_pairs, window_s
ize=1, negative=5),
    'NetMF(W2,N1)': embedding_builder('NetMF', init_para=netmf_pairs, window_s
ize=2, negative=1),
    'NetMF(W2,N5)': embedding_builder('NetMF', init_para=netmf_pairs, window_s
ize=2, negative=5),
    'NetMF(W3,N1)': embedding_builder('NetMF', init_para=netmf_pairs, window_s
ize=3, negative=1),
    'NetMF(W3,N5)': embedding_builder('NetMF', init_para=netmf_pairs, window_s
ize=3, negative=5),
    'NetMF(W5,N1)': embedding_builder('NetMF', init_para=netmf_pairs, window_s
ize=5, negative=1),
    'NetMF(W5,N5)': embedding_builder('NetMF', init_para=netmf_pairs, window_s
ize=5, negative=5),
#    'NetMF(W10,N1)': embedding_builder('NetMF', init_para=netmf_pairs, window
_size=10, negative=1),
#    'NetMF(W10,N5)': embedding_builder('NetMF', init_para=netmf_pairs, window
_size=10, negative=5),

#    'GraRep(2,N5)': embedding_builder('GraRep', init_para=grarep_para, negati
ve=5), # GraRep behaves like NetMF(W1)
#    'HONEM':  embedding_builder('HONEM'),
}
```

In [11]:
```
FON_edges = list((start[-1],next_node) for start in gen_HON.source_paths_len1
for _, next_node, _ in gen_HON.transition_probs(start))
#FON_edges
```

debug builder.evaluate_pairs

Before running extensive simulations, examine the best case - no cross validation:

- true_probs contains all transition probabilities for the excluded_edges, but we want all of them (set excluded_edges=FON_edges)
- the embedding uses the original transition probabilities (gen_HON)
- The embedding approximates the 1-step transitions [NetMF(window_size=1) or Experiment (its SGD equivalent)] instead of using random walks
- After decoding the probabilities, keep only those connected by a FON link and re-normalize

The outcome is worse than I had hoped for, rendering the whole analysis useless.

In [12]:
```
def debug_evaluate_pairs(builder, dimension=128, title=None, only_FON_edges=True, FON_edges=FON_edges, decode_args=decode_args, limit=(0.00001,1)):
    true_probs = builder.get_true_probs(gen_HON, excluded_edges=FON_edges) # true_probs for all edges
    gen_XE = gen_HON # no cross validation
    res = builder.evaluate_pairs(true_probs, gen_XE, dimension, **decode_args)
    df = pd.DataFrame(res).set_index(['source','target'])

    df.plot.scatter('true_prob', 'pred_prob', alpha=0.01, logx=True, logy=True, xlim=limit, ylim=limit)
    if title is not None: plt.suptitle(title)
    ax = plt.gca()
    ax.plot(limit,limit, '-')
```

In [13]:
```
if verbose:
    %time debug_evaluate_pairs(embedding_builders['NetMF(W1,N1)'], 128, 'HON NetMF(W=1,N=1,R=128)') # 3.5 sec
    plt.savefig('tmp/prob_netmf-w1n1r128.png')
    %time debug_evaluate_pairs(embedding_builders['NetMF(W3,N5)'], 16, 'HON NetMF(W=3,N=5,R=16)') # 11.5 sec
    plt.savefig('tmp/prob_netmf-w3n5r16.png')
    #%time debug_evaluate_pairs(embedding_builders['NetMF(W10,N5)'], 16, 'HON NetMF(W=10,N=5,R=16)') # 5 min
    #plt.savefig('tmp/prob_netmf-w310n5r16.png')

    #%time debug_evaluate_pairs(embedding_builders['NetMF(W10,N5)'], 256, 'NetMF(W10,N5)')
    #£%time debug_evaluate_pairs(embedding_builder('Experiment', dict(seed=1, neg_stationary=True), steps=1000, negative=1), 256, 'Experiment')
```

In [14]:
```
#for name,builder in embedding_builders.items():
#    %time debug_evaluate_pairs(builder, 16, name)
```

Crossvalidation of probability prediction for single embedding

Compares true and predicted probabilities using cross validation.

Cross validation means, that for a set of edges, a new instance `gen_XE` of CrossValidation_HigherOrderPathGenerator is generated, where all information about the transition probabilities along these edges is hidden. An embedding is trained using `gen_XE` and the (predicted) transition probabilities are decoded from this embedding.

In [15]:
```
from sklearn.model_selection import KFold
def evaluate_pairs(gen, builder, builder_id, dimension=128, n_splits=20, use_F
ON=False, plot=False, decode_args=decode_args, limit=(0.00001,1)):
    gen_build = gen
    if use_FON:
        gen_build = HigherOrderPathGenerator(node_sort_key=gen._node_sort_key,
id=gen._id + ' (FON)')
        for key in gen.rule_keys:
            if len(key)>1:
                continue
            for start, next_node,prob in gen.transition_probs(key):
                gen_build.add_rule(start, next_node, prob)
    res = []
    kf = KFold(n_splits=n_splits, shuffle=True, random_state=0)
    progress = tqdm(range(kf.n_splits))
    progress_iter = iter(progress)
    for i_split, (_, test_index) in enumerate(kf.split(FON_edges)):
        next(progress_iter)
        #progress.set_postfix(dict(split=i_split))
        excluded_edges = list(e for i,e in enumerate(FON_edges) if i in test_i
ndex) # = FON_edges[test_index]
        true_probs = embedding_builder.get_true_probs(gen, excluded_edges)
        gen_XE = CrossValidation_HigherOrderPathGenerator(gen_build, excluded_
edges, '%s exclude_%d' % (gen_build._id,i_split))
        res.extend(builder.evaluate_pairs(true_probs, gen_XE, dimension, **dec
ode_args))
    try:
        next(progress_iter)
    except StopIteration:
        pass
    res = pd.DataFrame(res).set_index(['source','target'])
    if plot:
        res.plot.scatter('true_prob', 'pred_prob', alpha=0.01, logx=True, logy
=True, xlim=limit, ylim=limit)
        ax = plt.gca()
        ax.plot(limit,limit, '-')
        plt.gcf().suptitle(('FON' if use_FON else 'HON') +' '+ builder_id + f'
CV(k={n_splits})')
    return res
```

In [16]:
```
if verbose:
    df = evaluate_pairs(gen_HON, embedding_builders['NetMF(W1,N1)'], 'NetMF(W=
1,N=1,R=128)', 128, n_splits=100, use_FON=False, plot=True)
    plt.savefig('tmp/cv-prob_netmf-w1n1r128.png')
```

In [17]:
```
if verbose:
    df2 = evaluate_pairs(gen_HON, embedding_builders['NetMF(W2,N1)'], 'NetMF(W
=2,N=1,R=16)', 16, n_splits=100, use_FON=False, plot=True)
    plt.savefig('tmp/cv-prob_netmf-w2n1r16.png')
```

In [18]:
```
if verbose:
    df2 = evaluate_pairs(gen_HON, embedding_builders['NetMF(W3,N5)'], 'NetMF(W
=3,N=5,R=16)', 16, n_splits=100, use_FON=False, plot=True)
    plt.savefig('tmp/cv-prob_netmf-w3n5r16.png')
```

Compare embeddings

In [19]:
```
from sklearn.model_selection import KFold
res = []
kf = KFold(n_splits=20, shuffle=True, random_state=0)
progress = tqdm(range(kf.n_splits * len(embedding_builders) * len(dimensions)
* 2))
progress_iter = iter(progress)
for i_split, (_, test_index) in enumerate(kf.split(FON_edges)):
    ##excluded_edges = FON_edges[test_index]
    excluded_edges = list(e for i,e in enumerate(FON_edges) if i in test_index
)
    true_probs = embedding_builder.get_true_probs(gen_HON, excluded_edges)
    for gen_name,gen in [('HON', gen_HON), ('FON', gen_FON)]:
        gen_XE = CrossValidation_HigherOrderPathGenerator(gen, excluded_edges,
'%s exclude_%d' % (gen_name,i_split))
        for builder_name, builder in embedding_builders.items():
            for dimension in dimensions:
                next(progress_iter)
                progress.set_postfix_str(f'{i_split}: {gen_name} {builder_name
} {dimension}')
                #progress.set_postfix(dict(split=i_split, gen=gen_name, model=
builder_name, dim=dimension))
                out = dict(model=gen_name, dimension=dimension, embedding=buil
der_name, split=i_split)
                out.update(builder.evaluate(true_probs, gen_XE, dimension, **d
ecode_args))
                res.append(out)
try:
    next(progress_iter)
except StopIteration:
    pass
df = pd.DataFrame(res)
df.head()
```

Out[19]:

	model	dimension	embedding	split	SSE	SSElog
0	HON	16	NetMF(W1,N1)	0	3.811116	27535.852668
1	HON	32	NetMF(W1,N1)	0	5.826772	29355.270917
2	HON	64	NetMF(W1,N1)	0	7.298458	31640.753643
3	HON	128	NetMF(W1,N1)	0	8.006961	33005.995388
4	HON	16	NetMF(W1,N5)	0	4.222847	28841.761619

In [20]:
```
df_sum = df.groupby(['model','dimension','embedding'])['SSE','SSElog'].sum()
df_sum.to_csv('tmp/cv-prob.csv', sep='\t', encoding='utf-16')
```

```
C:\Users\mstud\Anaconda3\envs\pathpy\lib\site-packages\ipykernel_launcher.py:
1: FutureWarning: Indexing with multiple keys (implicitly converted to a tupl
e of keys) will be deprecated, use a list instead.
  """Entry point for launching an IPython kernel.
```

In [21]:
```
df_sum['SSE'].sort_values()
```

Out[21]:
```
model  dimension  embedding
FON    16         NetMF(W5,N5)     47.792190
                  NetMF(W3,N5)     49.280430
                  NetMF(W5,N1)     49.350664
HON    16         NetMF(W5,N5)     49.384262
                  NetMF(W5,N1)     50.371449
                                      ...
       128        NetMF(W1,N5)     93.412453
       64         NetMF(W1,N1)    106.354252
FON    64         NetMF(W1,N1)    106.354252
HON    128        NetMF(W1,N1)    132.641770
FON    128        NetMF(W1,N1)    132.641770
Name: SSE, Length: 64, dtype: float64
```

In [22]:
```
df_sum['SSElog'].sort_values()
```

Out[22]:
```
model  dimension  embedding
HON    16         NetMF(W5,N1)    524060.605983
       32         NetMF(W5,N1)    526176.746898
       16         NetMF(W3,N1)    531328.282973
       32         NetMF(W3,N1)    538490.524515
       16         NetMF(W2,N1)    539406.553885
                                      ...
       128        NetMF(W1,N5)    636259.354630
FON    64         NetMF(W1,N1)    656262.486685
HON    64         NetMF(W1,N1)    656262.486685
       128        NetMF(W1,N1)    679515.663336
FON    128        NetMF(W1,N1)    679515.663336
Name: SSElog, Length: 64, dtype: float64
```

In [ ]:
```
```

Appendix D

Process Documentation

The following documents illustrate my plans for this thesis.

MSc Thesis: first ideas

According to the encoder-decoder framework of [Hamilton,Ying,Leskovec_2017b], the principal decision is whether a **shallow embedding** is learned, or whether a **neighbourhood aggregation** is used. At this early stage of the thesis, I cannot assess which approach works best. Hence, different ideas may be pursued in parallel for now.

Adaptation of a generic shallow embedding

Many of the baseline methods for creating shallow embeddings ([node2vec], [DeepWalk], [NetMF], [Honem], [Hope], [Line-2]) aim at finding a representation suitable to reconstruct **pairwise** (kind of) proximities – which corresponded to combinations of one- or more-step transition probabilities.

How are we going to generalize these pairwise probabilities to higher order networks?

Reasons for embedding higher-order (De Bruijn) memory nodes

For the first order network, the embedding Z_j of a node X_j is used to reconstruct P(X_j|X_{j-1}) from the scalar product <Z_j,Z_{j-1}> (and some transformation, e.g. softmax). Therefore, the natural **generalization to higher-order networks** is IMHO reconstructing P(X_j|X_{j-1},…,X_{j-k}) via the scalar product of <Z_j,Z_{j-1,…,j-k}> where the tuple (X_{j-1},…,X_{j-k}) is embedded as Z_{j-1,…,j-k}.

The transition probability P(X_j|X_{j-1},…,X_{j-k}) equals the corresponding edge probability in a k^{th} order De Bruijn Graph. Observing that the tuples of different lengths form a **hierarchy**, we evaluate the transition probabilities for different orders instead of using a fixed order k. Furthermore, all these transition probabilities form the basis for generating the null model for anomaly detection with [Hypa].

This hierarchy is also suitable to **regularize** embeddings for configurations lacking data, see [Hamilton,Ying,Leskovec_2017b, § 2.5.2 tying node embeddings across layers]. It also allows for curriculum learning.

Why (not) stick to pairwise proximities

Note that the term “High-order proximity” is used to describe pairwise proximities in [Hope].

A generalization might base on a similarity measure between two points, which somehow is based on paths through these two points.

However, the whole issue about bothering to use higher order models is the assumption that some transition probabilities are sensitive to steps of the random walk previous to the last one. Therefore, I guess that link prediction – apparently the most prominent task – based on pairwise similarities is doomed to fail. (No need to use higher order models to conclude that points within a community are more likely to be connected…)

Therefore, I had intended not to follow this approach.

However, I like the idea in [Hope] of breaking the symmetry by using **different embeddings for source and target nodes** of a directed edge.

Adaptation of GraphSAGE

[GraphSAGE] repeatedly aggregates features over a neighbourhood, but relies on random sampling instead of aggregating over the whole neighbourhood. This has the advantage of dealing with a fixed amount of data in each iteration.

However, the repeated aggregation inherently assumes a first order model (i.e. memoryless random walk). Hence, instead of k iterations of first order neighbourhoods, we should first aggregate over the neighbourhoods of each memory node in a k^{th} order De Bruijn graph, and repeat this for decreasing orders k. (Whether the second iteration uses $(k-1)^{th}$ order only or the union with k^{th} order needs to be decided.)

Memory

Since calculating the embedding builds on top of fitting a higher-order model to paths, we need to assume, that for the latter memory consumption is not prohibitive. However, I assume that the average degree of a higher-order (De Bruijn) memory node is much smaller than the dimension of the embedding (e.g. 128). Therefore, storing all the embedded values (assuming shallow embedding) could be much costlier (e.g. 50 times) than storing the transition probabilities in a sparse matrix.

The situation is similar for the higher-order adaption of GraphSAGE.

Therefore, there is no clear winner in terms of memory requirement.

Von: Michael Markus Studer/at/UZH
An: perri@ifi.uzh.ch
Kopie: scholtes@ifi.uzh.ch

Datum: Freitag, 24. Juli 2020 20:03
Betreff: MSc Thesis Michael

Dear Ingo
Dear Vincenzo

@Ingo: summary of the summary - embedding lattice with slow-down/speed-up successful; [InfiniteWalk] is limit of [NetMF] (which approximates [DeepWalk]) for window size to infinity and creates embedding from Laplacian, which might be applied to higher-order De Bruijn graph; Progress: code & new papers.

In the meeting of July 16, Ingo mentioned, that I might discuss with you (Vincenzo) some topics of my thesis. Not sure, what Ingo already told you about and whether you indeed need know my thoughts explained below...

Since I have been reading into the topic for some time, I thought that I might give you some summary about my ideas. This will hopefully simplify the discussion.

You probably already know that I am investigating embeddings for higher-order networks (i.e. higher-order markovian).
As far as transition probabilities in higher-order networks are concerned, I will build on top of MON, MoGEN, or [BuildHON+].

* Example network using non-markovian lattice
As mentioned in [Slow-down/speed-up], the higher-order terms might affect the diffusion speed, and therefore Ingo suggested investigating a two dimensional lattice where horizontal transitions are encouraged (e.g. moving right after moving right gets a higher probability at the expense of moving left after moving right) and vertical transitions are discouraged (e.g. after moving up, moving down is preferred over moving up again).

* Skip-gram with negative sampling (SGNS, aka Word2Vec) model
So far, I have focused on SGNS and therefore adjusted the random walk in [DeepWalk] to reflect the higher-order dynamics. As expected, the embeddings of the above lattice network reveals, that the nodes connected by a horizontal edge are closer then those connected by a vertical edge.
[LevyGoldberg] demonstrate, that SGNS corresponds to a matrix factorization (via SVD) of the logarithmic shifted positive pointwise mutual information (PMI), which inspired another embedding [NetMF]. (Which indeed confirms the effects of speed-up and slow-down on the embeddings by DeepWalk.) But we have to keep in mind, that the result of [LevyGoldberg] holds only if the embedding allows for perfect reconstruction.
[InfiniteWalk] covers the case of DeepWalk with window size tending to infinity and its connection to the Laplacian.

* Directed graphs
These embeddings all assume undirected graphs, which simplifies the calculation of multi-step probabilities but is generally not fulfilled. In a higher-order model, these multi-step probabilities

anyway have to be replaced - no big deal. Unfortunately, Word2Vec translates paths into word-context pairs by treating paths essentially as bidirectional. Hence, I will have to move on from using simple black boxes (Word2Vec and SVD) to more general methods of optimization (SGD).
Concerning directed graphs: I intend to borrow from [HOPE] the idea of having different embeddings for source and target nodes.

* Beyond pairwise interactions
For a first-order model, it is fine to consider embeddings, which optimize a loss function depending on pairwise interactions of nodes only. However, there is no way to to reconstruct the distribution of paths of length 3 from two-dimensional marginal distributions. In order to reconstruct/approximate transition probabilities from an embedding, I will need estimates of P(S_k|S_0,...,S_{k-1}) and not just P(S_k|S_0) for random walk [S_0,...,S_k].
Speaking in terms of encoders and decoders [EncDec], I consider two approaches for approximating P(S_k|S_0,...,S_{k-1}):
1) DEC(ENC_source(S_0->...->S_{k-1}), ENC_target(S_k))
2) DEC(ENC_0(S_0), ..., ENC_{k-1}(S_{k-1}), ENC_target(S_k))
For Approach 1, the decoder combines the two encodings with a scalar product (and a soft-max), which might be generalized for approach 2 to a higher-order inner product (used in [HOSGNS]), which is solved by tensor factorization.

Approach 1 still fits into the PMI framework and can be solved by SVD. For fully connected graphs, the number of paths to embed individually grows exponentially with the length. (But so do MON & MoGEN...)
This might be addressed by the fact that the subpaths form a hierarchy and that P(S_k|S_1,...,S_{k-1}; S_0 exists) = sum_x P(S_k|S_0=x,S_1,...,S_{k-1}) * P(S_0=x|S_1,...,S_{k-1})
i.e. P(S_k|S_1,...,S_{k-1}; S_0 exists) is a weighted average of P(S_k|S_0,S_1,...,S_{k-1}).
[EncDec, equation (18)] suggests regularizing penalties accross layers. Keeping this in mind, I intend to get some experience with SGD too.

Related to approach 1, I could apply [InfiniteWalk] to the maximal-order De Bruijn graph of the higher-order model:
DEC(ENC_source(S_0->...->S_{k-1}), ENC_target(S_1->...->S_k))
But I will have to see how to replace ENC_target(S_1->...->S_k) by ENC_target(S_k) and I will also need to dig into the proofs to see, whether they hold for directed graphs too.

* Experiments
So far, I did not spend much time thinking about experiments. I primarily had the reconstruction of the transition matrix in mind; maybe predicting the top 10 links, etc. However, having already trained embedding vectors for each subpath (in approach 1) means, that the first-order topology of the network is already known.

It is hard to tell, whether an embedding like ENC_source(S_0->...->S_{k-1}) is still useful for other tasks.

Most likely, I will seek your advice about experiments in future discussions.

Best,
Michael

References ([LevyGoldberg] and [EncDec] are key to understanding):
[BuildHON+] https://arxiv.org/abs/1712.09658
[Slow-down/speed-up] https://www.nature.com/articles/ncomms6024/
[DeepWalk] https://dl.acm.org/doi/pdf/10.1145/2623330.2623732
[LevyGoldberg] http://papers.nips.cc/paper/5477-neural-word-embedding-as
[NetMF] https://dl.acm.org/doi/pdf/10.1145/3159652.3159706

[InfiniteWalk] https://arxiv.org/abs/2006.00094
[HOPE] https://dl.acm.org/doi/pdf/10.1145/2939672.2939751
[EncDec] https://arxiv.org/abs/1709.05584
[HOSGNS] https://arxiv.org/abs/2006.14330

Von: Michael Markus Studer/at/UZH
An: scholtes@ifi.uzh.ch, perri@ifi.uzh.ch

Datum: Samstag, 05. September 2020 00:23
Betreff: Status MSc Thesis Michael Studer

Dear Ingo
Dear Vincenzo

Since I managed to implement the analyses discussed with Vincenzo one week ago, I would like to provide some status update. Some PDFs displaying Jupyter notebooks are also added to provide detailed information.

Examining the various papers on embeddings revealed the following tasks (starting with the most important one):
1) Classification
2) Link prediction
3) Graph reconstruction
4) Visualization
...
While HONEM (my direct competitor) covers all these four tasks, there is an unanimously preference for **multi-label classification** by DeepWalk, Node2vec, and NetMF (my primary inspiration).

Classification task
Following Vincenzo's advice, I decided, that the following datasets from SocioPatterns are best suited for a **multi-class classification** task:
- Contacts in a workplace (2013)
- Primary school temporal network data
- Hospital ward dynamic contact network
Using pathpy2 and the kdd2018-tutorial, I extracted paths, estimated the order (which happens to be 2 for all datasets), and exported the transition probabilities of order 1 and 2, see the attached SocioPatterns.pdf.

Next follows calculating various embeddings for visualization and classification learning - the latter calculates also score, F1-micro, F1-macro, confusion matrix, and auc roc. The models for classification were Logistic Regression, SVC, and Random Forest (20 and 100 estimators) - basically what came to my mind first. The following embeddings were used: HON NetMF (window_size 2 and 10), FON NetMF (window_size 2 and 10), HON DeepWalk (window_size 10), FON DeepWalk (window_size 10), and HONEM. In order to investigate the effect of using a HON vs a FON, I choose to include both a FON and a HON version of each embedding. Consequently, I excluded Node2vec as this would require a HON variant too - but this is perfectly feasible by manipulating the transition probabilities before using DeepWalk.

I have to admit, that I did not perform an exhaustive search over many parameters, and the results are rather a PoC than what is expected for the thesis.

The results were encouraging for the primary school dataset (see attached Classification_sim_v1_primaryschool.pdf), and I calculated the following:
- Visualizations (output 19) are ok
- F1-micro (output 25/26) and F1-macro (output 27/28) reveals that HON performs better than FON (averages over 10 simulations).

- There is also a confusion matrix (output 31) - for 1 simulation only, hence not directly comparable to F1 scores. (I might do this later.)
- For AUC (output 33), I aggregated pairs of true labels and decision scores over 10 simulations with 5 splits each and calculated the roc curves, using a one-vs-rest "approximation" (i.e. I hope this is correct).
However, for the workplace dataset, *HON performs worse* (see Classification_sim_v1_workplace.pdf).
Finally, for the hospital ward data, the F1 scores are generally lower than for the other two datasets and there is a tie between FON and HON (see Classification_sim_v1_hospital.pdf).

Link prediction (or rather probability prediction) task
I also tried to predict transition probabilities using a **cross validation** approach, where I carefully removed any information (besides its existence) about the probabilities of following a link by tweaking the transition probabilities (in CrossValidation_HigherOrderPathGenerator, which is not shown tough).
I had intended to do a grid search to identify parameters of (HON or FON) NetMF, which are best suited for this task.

Before actually running the simulation, I stepped back and examined, how well the probability reconstruction from the embedding actually works in the **best case** (i.e. no cross validation and even utilizing knowledge about network topology), see end of page 5 of the attached ProbabilityPrediction_sim.pdf. The scatter plots (page 7) where quite **disappointing**, causing me to consider this task a failure. May be datasets from the MON paper (when is a network a network?) - AIR, WIKI, TUBE - work better.

Conclusion
I think, the classification task is both relevant and convincing.
However, since the link prediction failed, I still have no example supporting my initial hypothesis, that for HON embeddings, pairwise decoding is not enough. (In fact, the findings using synthetic data do explain the benefit of going beyond pairwise.)

Unless instructed otherwise, I will focus on writing the thesis until our next meeting.

Best,
Michael

Anhänge:

SocioPatterns.pdf | Classification_sim_v1-primaryschool.pdf | Classification_sim_v1-workplace.pdf | Classification_sim_v1-hospital.pdf | ProbabilityPrediction_sim.pdf

Modularity calculation

Calculating the modularity of a higher order network may take a long time - especially when looping over all the pairs of nodes. Observe, that for a network n first-order nodes (each with constant degrees d), the number of k-th order nodes is $n \cdot d^{order-1}$. Therefore, the adjacency matrix is very sparse and its number of non-zero entries is only $n \cdot d^{order}$. Hence, it is crucial to avoid iterating over all $n^2 \cdot d^{2order-2}$ pairs of nodes.

The modularity q compares the number of actual edges connecting nodes of the same class with its expected counterpart. To count the actual edges, we need only to iterate over the non-zero entries of the adjacency matrix. While the term containing the expected number of edges apparently needs a loop over all pairs of nodes, its special structure (outer product) enables a faster calculation:
$\sum_{i,j} a[i] \cdot b[j] \cdot 1_{C[i]=C[j]} = \sum_c (\sum_i a[i] \cdot 1_{C[i]=c})(\sum_j b[j] \cdot 1_{C[j]=c})$

Furthermore, the implementation supports generalization of modularity to

- directed networks, as in Leicht and Newman (2007), "Community structure in directed networks"
- weighted networks, as in Newman (2004), "Analysis of weighted networks"

In [1]:

```
# this workbook needs pathpy2 installed
%conda list pathpy
```

```
# packages in environment at C:\Users\mstud\Anaconda3\envs\pathpy2:
#
# Name                    Version                   Build  Channel
pathpy2                   2.2.0                    pypi_0    pypi
Note: you may need to restart the kernel to use updated packages.
```

In [2]:

```
import pathpy as pp
import numpy as np
from collections import defaultdict
```

In [3]:

```
def calc_q(net, C, weighted=False, **kwargs):
    nodes = list(net.nodes.keys())
    mat = net.adjacency_matrix(weighted=weighted, **kwargs) # scipy.sparse.csc.csc_matr
ix
    mat_sum = 0
    mat_sum_by_row_C = defaultdict(float)
    mat_sum_by_col_C = defaultdict(float)
    q = 0
    for c in range(mat.shape[1]):
        c_C = C[nodes[c]] # class of current column's node
        for ind in range(mat.indptr[c], mat.indptr[c+1]):
            r = mat.indices[ind]
            v = mat.data[ind]
            # assert mat[r,c]==v
            r_C = C[nodes[r]] # class of current rows's node
            mat_sum += v
            mat_sum_by_row_C[r_C] += v
            mat_sum_by_col_C[c_C] += v
            if c_C == r_C:
                q += v
    q_exp = sum( v*mat_sum_by_col_C[c] for c,v in mat_sum_by_row_C.items() ) / (mat_sum
**2)
    q = q/mat_sum - q_exp
    q_max = 1 - q_exp
    return (q, q_max)
```

calc_q() still uses for loops. Consider converting with .tocoo(), which represents the matrix with three arrays (row, col, data), and use numpy array operations.

Some testdata

to compare with existing implementation

In [4]:

```
n = 100
net = pp.algorithms.random_graphs.erdoes_renyi_gnp(n, 0.2, self_loops=False, directed=F
alse)
for e in net.edges:
    net.edges[e]['weight'] = np.random.exponential()
net_D = pp.algorithms.random_graphs.erdoes_renyi_gnp(n, 0.2, self_loops=False, directed
=True)
for e in net_D.edges:
    net_D.edges[e]['weight'] = np.random.exponential()
```

In [5]:

```
classes = { str(i): 'group %d' % (i%5) for i in range(n) }
```

In [6]:

```
q = pp.algorithms.modularity.q(net, C=classes)
q_max = pp.algorithms.modularity.q_max(net, C=classes)
if q_max < 0:
    q_max+= 1
print(q,q_max)
```

```
-0.016743908659001307 0.7998893578740652
```

In [7]:

```
calc_q(net, classes, transposed=True) # weighted=False
```

Out[7]:

```
(-0.016743908659001383, 0.7998893578740648)
```

Calculation works for all variants of adjacency matrices (weighted and/or directed):

In [8]:

```
calc_q(net, classes, weighted=True, transposed=True)
```

Out[8]:

```
(-0.01852957868856711, 0.798714644094694)
```

In [9]:

```
calc_q(net_D, classes, weighted=True, transposed=True)
```

Out[9]:

```
(0.010614981011942137, 0.8010128521279705)
```

In [10]:

```
# calculation is invariant under transposition of the matrix
calc_q(net_D, classes, weighted=True, transposed=False)
```

Out[10]:

```
(0.010614981011942137, 0.8010128521279705)
```

In [11]:

```
# works also for HigherOrderNetwork.adjacency_matrix(include_subpaths, weighted, transp
osed)
# calc_q(hon, classes, weighted=True, include_subpaths=True)
```

In [ ]:

```

```

Nomenclature

Network

A Adjacency matrix

D Degree matrix

d_u Degree of node u

$\mathbf{D}^k_{\mathbf{neighbor}}$ k^{th}–order neighborhood matrix (HONEM)

$\mathcal{E}$ Set of edges

$\mathcal{E}_R, \mathcal{E}_U$ Rightward/horizontal and upward/vertical edges (lattice 2D)

$\mathcal{E}_{2R}, \mathcal{E}_{2U}$ Rightward and upward two–step edges (lattice 2D)

K Max order of Markov model

k Number of folds

L Laplacian matrix

L, R, U, D Directions (HON Lattice 2D)

$\overrightarrow{S_{1..L}}$ Random walk (abbreviation for $S_1 \to \ldots \to S_L$)

$T_{s \to t}$ First–order transition probability

$T^{(k)}_{s_{1..k} \to t}$ k^{th}–order transition probability

$\mathcal{V}$ Set of nodes

π Stationary distribution

ω Parameter for HON lattice 2D controlling the higher–order dynamic

Skipgram model and related embeddings

p, q Bias parameters (Node2vec)

$p(w, c)$ Distribution of word–context pairs (skip–gram model)

$p(w)$ Marginal distribution of $p(w, c)$ (skip–gram model)

$p(c|w)$ Conditional distribution of context given the word (skip–gram model)

$\hat{p}(c|w)$ Approximation $\langle \overrightarrow{w}, \overrightarrow{c} \rangle / const(w)$ of $p(c|w)$ based on embeddings (skip–gram model)

$\overrightarrow{w}, \overrightarrow{c}$ Embedding of word w and context c (skip–gram model)

$p_N(c)$ Distribution of the context for negative samples (skip–gram model)

$PMI(w, c)$ Pointwise mutual information ($p(c|w)/p_N(c)$) (skip–gram model)

L Either last index or length of a walk.

$L_{\max}$ Maximal size of a tuple for the consistency of walk probabilities

$\mathcal{L}$ Loss function

N Number of negative samples

R Dimension of embedding space

S Number of steps (GraRep)

τ Penalty parameter

W Window size

Indices

ℓ $\ell \leq L$ or $\ell \leq k$

w, c Word and context (skip–gram model)

u, v nodes of a network (nodes of a nedge); also De Bruijn nodes

s, t Source/start and target (transition probability)

List of Figures

List of Tables

List of Algorithms

Bibliography

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

[2] Y. Bengio, A. Courville, and P. Vincent. Representation Learning: A Review and New Perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(8):1798–1828, 2013.

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

[4] Shaosheng Cao, Wei Lu, and Qiongkai Xu. Deep neural networks for learning graph representations. In *AAAI*, volume 16, pages 1145–1152, 2016.

[5] Sudhanshu Chanpuriya and Cameron Musco. InfiniteWalk: Deep Network Embeddings as Laplacian Embeddings with a Nonlinearity, 2020.

[6] Karl Pearson F.R.S. LIII. On lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 2(11):559–572, 1901. doi: 10.1080/14786440109462720. URL https://doi.org/10.1080/14786440109462720.

[7] Valerio Gemmetto, Alain Barrat, and Ciro Cattuto. Mitigation of infectious disease at school: targeted class closure vs school closure. *BMC infectious diseases*, 14(1):695, December 2014. ISSN 1471-2334. doi: 10.1186/PREACCEPT-6851518521414365. URL http://www.biomedcentral.com/1471-2334/14/3841.

[8] Yoav Goldberg and Omer Levy. word2vec Explained: deriving Mikolov et al.'s negative-sampling word-embedding method, 2014.

[9] Christoph Gote, Giona Casiraghi, Frank Schweitzer, and Ingo Scholtes. Mogen: A genreative multi–order model to predict variable length paths in networks. KDD '20, 2020.

[10] Aditya Grover and Jure Leskovec. Node2vec: Scalable Feature Learning for Networks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 855–864, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342322. doi: 10.1145/2939672.2939754. URL https://doi.org/10.1145/2939672.2939754.

[11] MATHIEU GÉNOIS, CHRISTIAN L. VESTERGAARD, JULIE FOURNET, ANDRÉ PANISSON, ISABELLE BONMARIN, and ALAIN BARRAT. Data on face-to-face contacts in an office building suggest a low-cost vaccination strategy based on community linkers. *Network Science*, 3:326–347, 9 2015. ISSN 2050-1250. doi: 10.1017/nws.2015.10. URL http://journals.cambridge.org/article_S2050124215000107.

[12] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in neural information processing systems*, pages 1024–1034, 2017.

[13] William L. Hamilton, Rex Ying, and Jure Leskovec. Representation Learning on Graphs: Methods and Applications, 2017.

[14] M. Khosla, V. Setty, and A. Anand. A Comparative Study for Unsupervised Network Representation Learning. *IEEE Transactions on Knowledge and Data Engineering*, pages 1–1, 2019.

[15] Tamara G. Kolda and Brett W. Bader. Tensor Decompositions and Applications. *SIAM Review*, 51(3):455–500, 2009. doi: 10.1137/07070111X. URL https://doi.org/10.1137/07070111X.

[16] Joseph B Kruskal. Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis. *Psychometrika*, 29(1):1–27, 1964.

[17] Renaud Lambiotte, Martin Rosvall, and Ingo Scholtes. From networks to optimal higher-order models of complex systems. *Nature physics*, 15(4): 313–320, 2019.

[18] E. A. Leicht and M. E. J. Newman. Community Structure in Directed Networks. *Phys. Rev. Lett.*, 100:118703, Mar 2008. doi: 10.1103/PhysRevLett.100.118703. URL https://link.aps.org/doi/10.1103/PhysRevLett.100.118703.

[19] Omer Levy and Yoav Goldberg. Neural Word Embedding as Implicit Matrix Factorization. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2177–2185. Curran Associates, Inc., 2014. URL http://papers.nips.cc/paper/5477-neural-word-embedding-as-implicit-matrix-factorization.pdf.

[20] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient Estimation of Word Representations in Vector Space, 2013.

[21] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed Representations of Words and Phrases and their Compositionality. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 3111–3119. Curran Associates, Inc., 2013. URL `http://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases`.

[22] M. E. J. Newman. Analysis of weighted networks. *Phys. Rev. E*, 70:056131, Nov 2004. doi: 10.1103/PhysRevE.70.056131. URL `https://link.aps.org/doi/10.1103/PhysRevE.70.056131`.

[23] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710, 2014.

[24] Simone Piaggesi and André Panisson. Time-varying Graph Representation Learning via Higher-Order Skip-Gram with Negative Sampling, 2020.

[25] Jiezhong Qiu, Yuxiao Dong, Hao Ma, Jian Li, Kuansan Wang, and Jie Tang. Network Embedding as Matrix Factorization: Unifying DeepWalk, LINE, PTE, and Node2vec. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*, WSDM '18, page 459–467, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355810. doi: 10.1145/3159652.3159706. URL `https://doi.org/10.1145/3159652.3159706`.

[26] Radim Rehurek. Deep learning with word2vec and gensim, 2013. URL `https://rare-technologies.com/deep-learning-with-word2vec/`. [Online; accessed 2020-11-15].

[27] Martin Rosvall, Alcides V Esquivel, Andrea Lancichinetti, Jevin D West, and Renaud Lambiotte. Memory in network flows and its effects on spreading dynamics and community detection. *Nature communications*, 5(1):1–13, 2014.

[28] Mandana Saebi, Giovanni Luca Ciampaglia, Lance M Kaplan, and Nitesh V Chawla. HONEM: Network Embedding Using Higher-Order Patterns in Sequential Data, 2019.

[29] Ingo Scholtes. When is a Network a Network? Multi-Order Graphical Model Selection in Pathways and Temporal Networks. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, page 1037–1046, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450348874. doi: 10.1145/3097983.3098145. URL `https://doi.org/10.1145/3097983.3098145`.

[30] Ingo Scholtes, Nicolas Wider, René Pfitzner, Antonios Garas, Claudio J Tessone, and Frank Schweitzer. Causality-driven slow-down and speed-up of diffusion in non-Markovian temporal networks. *Nature communications*, 5(1):1–9, 2014.

[31] Juliette Stehlé, Nicolas Voirin, Alain Barrat, Ciro Cattuto, Lorenzo Isella, Jean-François Pinton, Marco Quaggiotto, Wouter Van den Broeck, Corinne Régis, Bruno Lina, and Philippe Vanhems. High-Resolution Measurements of Face-to-Face Contact Patterns in a Primary School. *PLOS ONE*, 6(8): e23176, 08 2011. doi: 10.1371/journal.pone.0023176. URL `http://dx.doi.org/10.1371/journal.pone.0023176`.

[32] Jian Tang, Meng Qu, and Qiaozhu Mei. PTE: Predictive Text Embedding through Large-Scale Heterogeneous Text Networks. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '15, page 1165–1174, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336642. doi: 10.1145/2783258.2783307. URL `https://doi.org/10.1145/2783258.2783307`.

[33] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. LINE: Large-Scale Information Network Embedding. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15, page 1067–1077, Republic and Canton of Geneva, CHE, 2015. International World Wide Web Conferences Steering Committee. ISBN 9781450334693. doi: 10.1145/2736277.2741093. URL `https://doi.org/10.1145/2736277.2741093`.

[34] Leo Torres, Kevin S Chan, and Tina Eliassi-Rad. GLEE: Geometric Laplacian Eigenmap Embedding. *Journal of Complex Networks*, 8(2), 03 2020. ISSN 2051-1329. doi: 10.1093/comnet/cnaa007. URL `https://doi.org/10.1093/comnet/cnaa007`. cnaa007.

[35] Anton Tsitsulin, Davide Mottin, Panagiotis Karras, and Emmanuel Müller. VERSE: Versatile Graph Embeddings from Similarity Measures. In *Proceedings of the 2018 World Wide Web Conference*, WWW '18, page 539–548, Republic and Canton of Geneva, CHE, 2018. International World Wide Web Conferences Steering Committee. ISBN 9781450356398. doi: 10.1145/3178876.3186120. URL `https://doi.org/10.1145/3178876.3186120`.

[36] Philippe Vanhems, Alain Barrat, Ciro Cattuto, Jean-François Pinton, Nagham Khanafer, Corinne Régis, Byeul-a Kim, Brigitte Comte, and Nicolas Voirin. Estimating Potential Infection Transmission Routes in Hospital Wards Using Wearable Proximity Sensors. *PLoS ONE*, 8(9):e73970, 09 2013. doi: 10.1371/journal.pone.0073970. URL `http://dx.doi.org/10.1371%2Fjournal.pone.0073970`.

[37] Daixin Wang, Peng Cui, and Wenwu Zhu. Structural Deep Network Embedding. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 1225–1234, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342322. doi: 10.1145/2939672.2939753. URL `https://doi.org/10.1145/2939672.2939753`.

[38] Jian Xu, Thanuka L. Wickramarathne, and Nitesh V. Chawla. Representing higher-order dependencies in networks. *Science Advances*, 2(5), 2016. doi: 10.1126/sciadv.1600028. URL `https://advances.sciencemag.org/content/2/5/e1600028`.

[39] Jian Xu, Mandana Saebi, Bruno Ribeiro, Lance M. Kaplan, and Nitesh V. Chawla. Detecting Anomalies in Sequential Data with Higher-order Networks, 2017.

[40] D. Zhang, J. Yin, X. Zhu, and C. Zhang. Network Representation Learning: A Survey. *IEEE Transactions on Big Data*, 6(1):3–28, 2020.

[41] Chang Zhou, Yuqiong Liu, Xiaofei Liu, Zhongyi Liu, and Jun Gao. Scalable graph embedding for asymmetric proximity. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.