Master's Thesis

M.Sc. Business Administration

University of Hamburg
Faculty of Business Administration
Chair of Statistics with Application in Business Administration
Prof. Dr. Martin Spindler

Analyse von Netzwerkdaten mittels Graph Representation Learning

Examiner: Prof. Dr. Martin Spindler

Submitted by:

Benjamin-Vincent Wankelmuth

6961405

Hagemannsweg 10a

21339 Lüneburg

017623998530

b.wankelmuth@gmx.de

Supervised by:

Prof. Dr. Martin Spindler

Submission date, place:

30/07/22, Hamburg

Abstract

In recent years, graphs have emerged as important tools for representing a wide range of systems, from social networks to proteins as biological structures. At the same time, this has led to an increasing need for methods to solve problems that involve graphs. Neither standard machine learning methods nor deep learning methods such as convolutional neural networks are able to learn a graph structure and then use it to make predictions. For this reason, graph neural networks (GNNs) have been developed that can successively learn the graph structure and incorporate it into predictions. The potential of GNNs to outperform previous deep learning methods has already been demonstrated on a variety of benchmark datasets such as Cora and CiteSeer. However, the performance of GNNs has not yet been verified on real enterprise data.

This gap is closed in this master's thesis. Within the framework of a research project with the pharmaceutical company Novartis, we investigate on the basis of suitable metrics how the performance of GNNs compares to the performance of a selection of suitable baselines using company data for the three learning tasks of node classification, link prediction and time series analysis. For this purpose, we introduce all required foundations of graphs and expound all GNNs used in the research project. Finally, after a comprehensive documentation of the research project, we elaborate the main findings and propose possible research directions.

By and large, our results corroborate the prevailing finding in the literature that GNNs are a powerful tool to generate better forecasts based on learned graph structures. For link prediction and time series forecasting, we observe that the best performance is achieved by a GNN. In this context, we also conclude that especially for time series forecasting the full potential of the GNNs could not be fully exploited due to certain deficiencies in the underlying data and therefore a better performance can be expected without these deficiencies in the data. In node classification, however, the performance of the best GNN lags behind the performance of the best baseline. Two GNNs even show a performance that would be expected with random guessing. From this we conclude that in this case the graph has little or no explanatory power for the target variable in question.

Contents

List of Figures

List of Tables

List of Abbreviations

AA	Adamic-Adar
ACC	Accuracy
AP	Average Precision
API	Application Programming Interface
AUC	Area Under the Curve
BC	Breast Cancer
BCE	Binary Cross Entropy
CHEB	Graph Neural Network with Chebyshev Convolution
CN	Common Neighbors
CNN	Convolutional Neural Network
CSS	Cascading Style Sheets
DCGRU	Diffusion Convolutional Gated Recurrent Unit
DCRNN	Diffusion Convolutional Recurrent Neural Network
DGCNN	Dynamic Graph Convolutional Neural Network
DRNL	Double Radius Node Labeling
DW	DeepWalk
GAE	Graph Autoencoder
GAT	Graph Attention Network
GCN	Graph Convolutional Network
GNN	Graph Neural Network
HCP	Health Care Professional
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
JC	Jaccard
KA	Katz
KL	Kullback-Leibler
LRC	Logistic Regression
MAE	Mean Absolute Error
MAPE	Mean Absolute Percentage Error
MLP	Multilayer Perceptron
NV	node2vec
PA	Preferential Attachment
RA	Resource Allocation

RFC	Random Forest Classifier
RMSE	Root Mean Square Error
RNN	Recurrent Neural Network
RPR	Rooted PageRank
SEAL	Learning from Subgraphs, Embeddings, and Attributes for Link Prediction
SC	Spectral Clustering
SR	SimRank
SVC	Support Vector Classifier
t-SNE	t-Distributed Stochastic Neighbor Embedding
XML	Extensible Markup Language
VAR	Vector Autoregression
VGAE	Variational Graph Autoencoder

List of Symbols

A	Adjacency matrix
$\mathcal{E}$	Set of edges
$\mathcal{G}$	Graph
(u, v)	Edge from node u to node v
(u, τ, v)	Edge from node u to node v of edge type τ
$\mathcal{V}$	Set of nodes
X	Node features
τ	Edge type

1 Introduction

Graphs are ubiquitous due to their ability to represent dependencies and interrelationships between entities (Waikhom and Patgiri (2021); Hamilton (2020)). They comprise a set of nodes and the respective dependencies between the pairs of nodes, which are represented by edges. As a universal language for describing complex systems, graphs have great expressive power (Liu and Zhou (2020)) and can thus be used to represent a wide variety of structures. For example, to encode a social network as a graph, nodes can be used to represent individuals and edges can be used to represent that there is a friendship between two individuals. Graphs are also suitable for biological applications: Here, nodes can represent proteins and edges the biological interactions between them (Hamilton (2020)).
The graph formalism has its particular strength first because of its focus on the relations between nodes rather than on the properties of the nodes themselves and second because of its generality. As we have seen above, the same graph formalism can represent different structures. Beyond a theoretical framework, graphs also provide a mathematical basis for analyzing, understanding and ultimately learning from complex systems (Hamilton (2020)).

Graph neural networks

Standard machine learning methods, such as logistic regression, can also be used for certain graph-related tasks, such as classification of nodes, but explicitly do not take the graph structure into account in their predictions. In cases where the graph structure is useful for predictions, standard machine learning methods cannot exploit this potential. Thus, we need effective and novel techniques to solve graph-based tasks leveraging the graph structure (Waikhom and Patgiri (2021)). The key to this is provided by graph neural networks (GNNs). They are based on the idea of learning representations for the nodes in the graph by repeatedly updating the representations of each node with the aggregated representations of the neighbors of that node. The resulting node representations can then be used as such or further aggregated into link representations or graph representations depending on the downstream task (Wu et al. (2022)). Building on this basic principle, a variety of GNNs have already been developed for different tasks. Among other things, these differ in how they aggregate and update node representations (Wu et al. (2021)). The advantages of GNNs include convincing performance as well as high interpretability (Liu and Zhou (2020)). Furthermore, GNNs can be applied in open source software like Python, where

even specific libraries are available for use.

Research objective

In the context of this master's thesis a research project with the pharmaceutical company Novartis is carried out. For this purpose we receive different company data from Novartis. An important part of this is data on sales figures of drugs for the treatment of breast cancer and melanoma as well as data on health care professionals (HCPs) who can prescribe the mentioned drugs. From the latter data, an HCP graph is created using additional data sources obtained through web scraping.
The project pursues four tasks. The first task, which is not GNN-related and thus somewhat detached from the other three tasks, is to determine influencers, i.e., the most important HCPs, on the basis of the HCP graph. However, the real focus of the project is on the remaining three tasks, which are all GNN-related. The second task is node classification, the third task is link prediction and the fourth task is time series forecasting. The overall goal is to compare the performance of different GNNs with the performance of a variety of baselines in these three learning tasks and thus determine whether the GNNs can outperform these baselines. For this we work with Python and ensure the reproducibility of our results wherever possible.[1]
Our application of GNNs to enterprise data is interesting because GNNs are usually applied to benchmark datasets in a research context. Our goal is to find out whether the potential that GNNs have demonstrated on benchmark datasets also translates to our data. This illustrates that the research project has a strong exploratory focus.
We track the performance of the GNNs and baselines in the three learning tasks using appropriate metrics to compare them. Visualizations are used extensively throughout the project to provide as much insight into the models as possible.

Overview

The remainder of the master's thesis is organized as follows: Section 2 reviews all the basics needed for further work with GNNs. For this purpose, the basics of graphs are introduced first and different machine learning tasks related to graphs are discussed. Furthermore, it is explained why standard machine learning methods are insufficient for graphs and why

[1] In exceptional cases, complete reproducibility of the results is not possible. This is the case for web scraping, for example. However, the resulting deviations would be minimal.

GNNs are needed. Subsequently, in section 3 all GNNs employed in the research project are elucidated. We first learn about the general functionality of a GNN, before we work out the required GNNs for each of our tasks. The theoretically acquired knowledge is deepened in section 4 in the context of the research project mentioned above. We work our way from data description, data processing and graph generation to our four learning tasks. The project is of vast extent, which also makes the documentation very extensive.[2]. Finally, in section 5, the main findings of the investigations are discussed and critically reflected upon, before this master's thesis is concluded with an outlook in section 6. Appendices A to F contain additional information and plots, especially concerning the research project. All notebooks belonging to the master's thesis are located in the folder `gnn_research_project` on the enclosed data carrier.

[2]The extensive documentation should make it possible to understand which part of the project is completed in which notebook. In addition, the decisions made in the course of the project, e.g., in data processing, should become understandable.

2 Basics

The following sections are rather technical and require a sound prior knowledge of graphs. Therefore, our aim in this section is to lay these foundations. We also address common machine learning tasks on graphs. Moreover, we motivate the need for GNNs with an example by showing why 2D convolution is not applicable to graphs, and how graph convolution solves this problem.

2.1 Foundations of graphs

There is a key terminological difference between the two common terms *graph* and *network*. While the term graph is conceptually dominant in machine learning to refer to the abstract data structure, the term network is primarily used in data mining and network science to describe specific instantiations of this data structure in the real world. This terminological distinction coincides with the distinction between *graph theory* and *network analysis*. The former studies the theoretical properties of the mathematical graph abstraction, whereas the latter deals with the properties of real-world data (Hamilton (2020)).[3]

A graph $\mathcal{G}$ is formally defined by a node set $\mathcal{V}$ and an edge set $\mathcal{E}$. While $\mathcal{V}$ contains all nodes existing in $\mathcal{G}$, $\mathcal{E}$ contains all edges existing between the nodes in $\mathcal{G}$. Here, an edge from node $u \in \mathcal{V}$ to node $v \in \mathcal{V}$ is denoted by $(u, v) \in \mathcal{E}$. In *simple graphs*, there is at most one edge between two nodes and no edges between a node and itself, with all edges being undirected, i.e., $(u, v) \in \mathcal{E} \leftrightarrow (v, u) \in \mathcal{E}$.
With the help of an *adjacency matrix* $A \in \mathbb{R}^{|\mathcal{V}| \times |\mathcal{V}|}$, a graph $\mathcal{G}$ can be represented. To do this, the nodes of the graph are ordered so that each node of the graph indexes a particular row and column. Then, the entries of A given by

$$A[u, v] = \begin{cases} 1, & \text{if } (u, v) \in \mathcal{E}, \\ 0, & \text{otherwise.} \end{cases}$$

can be used to indicate the existence of an edge. If the graph has only undirected edges, A is symmetric. In the case of a graph with *directed edges*, A is not necessarily symmetric. A graph can also have *weighted edges*, which allows the entries in the adjacency matrix to be arbitrarily real-valued and no longer restricted to $\{0, 1\}$. For example, a weighted edge in a graph may express the strength of the connection between two nodes.

[3] Since we are in machine learning and prefer a uniform term, we predominantly use the term graph throughout this thesis.

In addition to distinguishing between undirected, directed, and weighted edges, different *edge types* can also be considered. In a graph representing drug-drug interactions, different edges correspond to different side effects that occur when two drugs are taken simultaneously. Accordingly, the edge notation can be extended by an *edge type* or *relational type* τ to $(u, \tau, v) \in \mathcal{E}$. Similarly, a graph can also have different *node types*. Graphs considered so far that have only one node type and edge type are called *homogeneous graphs* (Elinas (2019)), whereas graphs with multiple edge types are called *multi-relational graphs*. For the latter, there is now an adjacency matrix A_τ for each edge type, such that they can be described by an adjacency tensor $\mathcal{A} \in \mathbb{R}^{|\mathcal{V}| \times |\mathcal{R}| \times |\mathcal{V}|}$, where $\mathcal{R}$ is the relation set (Hamilton (2020)).

Multi-relational graphs have two important subsets given by *heterogeneous graphs* and *multiplex graphs*.

In heterogeneous graphs, there are various node types based on which nodes can be partitioned into disjoint sets $\mathcal{V} = \mathcal{V}_1 \cup \mathcal{V}_2 \cup \cdots \cup \mathcal{V}_k$ with $\mathcal{V}_i \cap \mathcal{V}_j = \emptyset$, $\forall i \neq j$. The edges usually satisfy conditions with respect to node types, e.g., certain edges can exclusively connect nodes with different node types, i.e., $(u, \tau_i, v) \in \mathcal{E} \rightarrow u \in \mathcal{V}_j, v \in \mathcal{V}_k$. In a heterogeneous biomedical graph, let us consider three types of nodes, representing proteins, drugs, and diseases. Edges representing treatment can exist here only between drug nodes and disease nodes, while edges representing side effects arising from polypharmacy can exist only between two drug nodes. Heterogeneous graphs have the special case of *multipartite graphs*, where edges can exclusively connect nodes with different node types, i.e., $(u, \tau_i, v) \in \mathcal{E} \rightarrow u \in \mathcal{V}_j, v \in \mathcal{V}_k \wedge j \neq k$.

Multiplex graphs are based on the assumption that they can be decomposed into a set of k layers. Each node is associated with each layer and each layer has a single relation, which is called the *intra-layer* edge type of that layer. There may also be *inter-layer* edge types connecting the same node between different layers. In a multiplex transport network, each node represents a city and each layer represents a different mode of transport. In this case, the intra-layer edges represent the connections of cities through different transport means, while the inter-layer edges represent the possibility to change the transport means within a city (Hamilton (2020)).

Often there is *feature information* for a graph. This is mostly node-related attributes, which are stored in a real-valued matrix $X \in \mathbb{R}^{|\mathcal{V}| \times m}$, in which the nodes are ordered in

correspondence with the adjacency matrix. In heterogeneous graphs, it is usually assumed that each node type has its own attribute type. Rarely, graphs occur which have real-valued edge features in addition to discrete edge features, or where even real-valued features are assigned to the entire graph (HAMILTON (2020)).

2.2 Machine learning on graphs

Machine learning models are usually categorized depending on the task that they are meant to solve. In *supervised tasks*, the objective is to predict a target output given an input datapoint. By contrast, in *unsupervised tasks*, the objective is to infer patterns, such as clusters of points, in the data.
The boundaries between these two categories become blurry with machine learning problems on graphs. In the following, we present four very important and well-studied machine learning tasks on graph data. For these, we refer to HAMILTON (2020).

Node classification

The standard setting for node classification is that we are given the true labels on a training set of nodes $\mathcal{V}_{\text{train}} \subset \mathcal{V}$. The goal is to predict the label y_u, which could be a type, category, or feature.
For instance, node classification can be deployed to classify bots in a large social networking dataset with millions of users. As this task is impracticable manually, we would like to have a model capable of classifying users as a bot given only a small number of manually labeled examples. Apart from that, node classification is also suitable to classify the function of proteins in the interactome (HAMILTON ET AL. (2017)) and to classify the topic of documents based on hyperlink or citation graphs (KIPF AND WELLING (2017)).
Node classification is, however, not a simple variation of standard supervised classification and comes with significant differences. Most importantly, the nodes in a graph are not *independent and identically distributed (i.i.d)*. In supervised machine learning models, each datapoint is assumed statistically *independent* from all the other datapoints and the datapoints are assumed to be *identically distributed* so that the model can generalize to new datapoints. With node classification, this i.i.d. assumption is broken. We model an interconnected set of nodes rather than a set of i.i.d. datapoints.
By leveraging the connections between nodes, node classification models can become very successful. Common concepts to be used in node classification models include *homophily*,

the tendency for nodes to share features with their neighbors in the graph (McPherson et al. (2001), *structural equivalence*, the idea that nodes with similar local neighborhood structures have similar labels (Zhou et al. (2003)), as well as *heterophily*, which presumes that nodes are preferentially connected to nodes with different labels (Donnat et al. (2018)).[4]
These concepts can be exploited for modelling the relationship between nodes in node classification models.

With node classification models, we are given the full graph, including all the unlabeled nodes, during training. Hence, the only thing we do not have access to is the labels of the test nodes. Nevertheless, the knowledge of the test nodes' neighborhood in the graph can be used as information to improve our model during training. In this respect, node classification differs from the usual supervised setting, in which unlabeled datapoints are completely unobserved during training.
Node classification is often referred to as a *semi-supervised* (Yang et al. (2016)) task. This expression is based on the term semi-supervised learning, which is generally used for models that combine labeled and unlabeled data during training.[5]

Link prediction

In a standard setting for link prediction[6], we are given a set of nodes $\mathcal{V}$ and an incomplete set of edges between these nodes $\mathcal{E}_{\text{train}} \subset \mathcal{E}$. The goal is to use this partial information to infer the missing edges $\mathcal{E} \setminus \mathcal{E}_{\text{train}}$.
Real-world applications of link prediction include making content recommendations to users in social platforms (Ying et al. (2018)), predicting drug side-effects (Zitnik et al. (2018)), or inferring new facts in a relational database (Bordes et al. (2013)). A link prediction task's complexity depends crucially on the type of graph data in question. For example, in simple graphs, such as social networks that only encode "friendship" relations, simple heuristics based on the number of neighbors two nodes share can achieve strong performance (Lü and Zhou (2011)). However, in more complex multi-relational group datasets, such as biomedical knowledge graphs encoding hundreds of different biological in-

[4] For example, gender is a feature that exhibits heterophily in many social networks.
[5] It is important to note, however, that standard formulations of semi-supervised learning still require the i.i.d. assumption, which does not hold for node classification.
[6] Link prediction is also known as relation prediction, graph completion, and relational inference, depending on the specific application domain.

teractions, link prediction can require complex inference strategies (Nickel et al. (2016). Link prediction is also semi-supervised and requires inductive biases specific to the graph domain. The many variants of link prediction reach from settings where predictions are made over a single, fixed graph (Lü and Zhou (2011)) to settings where relations must be predicted across multiple disjoint graphs (Teru et al. (2020)).

Clustering and community detection

Since both node classification and link prediction aim to infer *missing* information about graph data, those two tasks can in many ways be seen as the graph analogs of supervised learning. In contrast, *community detection* is the graph analog of unsupervised learning.
Imagine we could make a *collaboration graph* that connects two researchers if they have co-authored a paper together using all citation information in Google Scholar. When examining this graph, we would be likely to obtain a graph that segregates into different *clusters* of nodes, grouped together by research area, institution, or other demographic factors. So, we would expect to observe a *community structure* in the network, where nodes are much more likely to be connected to nodes in the same community.
This is the core idea behind community detection, its challenge being to infer latent community structures given only the input graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. Community detection has many real-world applications, among which are uncovering functional modules in genetic interaction networks (Agrawal et al. (2018)) and uncovering fraudulent groups of users in financial transaction networks (Pandit et al. (2007)).

Graph classification, regression, and clustering

Machine learning applications on graph data are also popular in classification, regression, or clustering problems over entire graphs. For example, we might want to build a regression model capable of predicting a molecule's toxicity or solubility based on a graph representing its structure (Gilmer et al. (2017)). In another situation, we might want to build a classification model to detect whether a computer program is malicious by analyzing a graph-based representation of its syntax and data flow (Li et al. (2019c)). In such *graph classification* or *graph regression* applications, our goal is to learn over graph data. However, we are not interested in making predictions over the individual components of the single graph, i.e., the nodes or the edges. Instead, we aim to make independent predictions

specific to each graph given a dataset of *multiple different graphs*. In the related task of *graph clustering*, the objective is to learn an unsupervised measure of similarity between pairs of graphs.
Among all the machine learning tasks on graphs, graph regression and classification are likely the most obvious analogs of standard supervised learning. Here, each graph is an i.i.d. datapoint associated with a label, the goal being to use a labeled set of training points to learn a mapping from datapoints, i.e., graphs, to labels. In a similar fashion, graph clustering is the most obvious extension of unsupervised clustering for graph data. The fundamental challenge in these graph-level tasks is how to define useful features that consider the relational structure within each datapoint.

Besides these four very prominent machine learning tasks on graph data, there are other machine learning tasks that involve handling a graph. One of them is *time series forecasting*.

Time series forecasting

Multivariate time series forecasting has emerged as another noteworthy machine learning application on graph data. It is predominantly found in the discipline of traffic forecasting on road networks. For instance, we might be interested in predicting the future traffic speeds of multiple sensors based on historical traffic speeds of these sensors and an underlying road network (Li et al. (2018b)). In such a standard scenario, the graph is already given as a prior. However, recent research has revealed that the forecasting performance in multivariate time series forecasting can be improved by not giving a graph as prior, but letting the GNN learn the graph from the time series. With the approach of Cao et al. (2020), for example, inter-series correlations and temporal dependencies can be learned jointly in the spectral domain by the GNN in the form of a dependency graph. Shang et al. (2021) propose a related approach in which the pairwise information between multiple time series is exploited and learned by a graph, thereby enabling better forecasts.

2.3 Necessity of GNNs

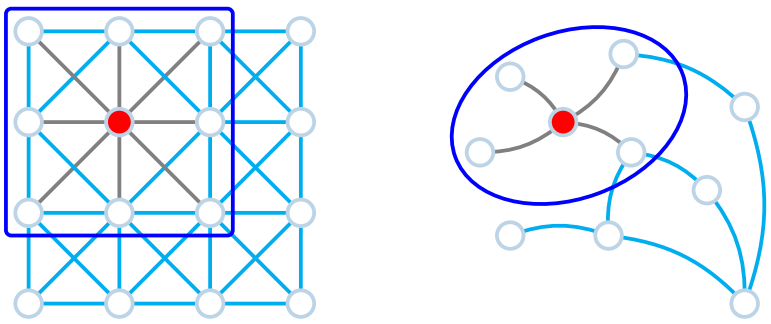
As we have already learned, standard machine learning methods as such are not applicable to graph data. Due to its complexity, graph data poses significant challenges to those methods mainly for two reasons.

1. Graphs can be irregular, i.e., they may have a variable size of unordered nodes, and

nodes from a graph may have a different number of neighbors, so that some important operations, e.g., convolutions, are easy to compute in the image domain, but difficult to apply in the graph domain.

2. Graphs do not fulfil the core assumption of existing machine learning algorithms that instances are independent of each other. This is because in graphs, each instance, i.e., each node, is related to others by edges of various types, such as citations, friendships, and interactions.

In order to make deep learning methods[7] applicable to graphs as well, new generalizations and definitons of important operations have been developed to handle the complexity of graph data, inspired by approaches such as CNNs, RNNs, and autoencoders. As an example, a graph convolution, the main building block of GNNs, can be generalized from a 2D convolution, the main building block of CNNs. The underlying idea is that an image can be considered as a special case of graphs where pixels are connected by adjacent pixels. Analogous to 2D convolution, graph convolutions can then be performed by taking the weighted average of a node's neighborhood information. Figure 1 (Wu et al. (2021)) juxtaposes a 2D convolution and a graph convolution, exemplifying the different approaches of the two operators.



(a) 2D convolution (b) Graph convolution

Figure 1: 2D convolution vs. graph convolution

With 2D convolution, each pixel in an image is taken as a node where neighbors are determined by the filter size. The 2D convolution then takes the weighted average of pixel values of the red node along with its neighbors. The neighbors of a node are ordered and have fixed size.

By contrast, to get a hidden representation of the red node with graph convolution, one

[7] Deep learning methods form part of machine learning methods.

simple solution of the graph convolutional operator is to take the average value of the node features of the red node along with its neighbors. Different from image data, the neighbors of a node are unordered and variable in size (Wu et al. (2021)).

Ultimately, we can see that 2D convolution as an operator from standard machine learning cannot be applied to graphs, because it cannot deal with the unordered neighborhood structure and also assumes that the nodes are independent, which is not the case for a graph. However, as the figure also bears out, graph convolution can adapt to different neighborhood structures and thus also capture the dependencies between the nodes. This substantiates the necessity of GNNs, which we cover in detail in the next section.

3 Graph neural networks in theory

After acquiring all relevant prior knowledge in the previous section, we now introduce the GNNs for our three learning tasks of node classification, link prediction and time series forecasting, which are applied in the research project.
For this purpose, we choose the following procedure: First, we give a general framework of GNNs and outline how a GNN is structured. We also specify the notation used throughout this section. Subsequently, we use this framework to introduce the relevant GNNs for our first task, node classification. This is followed by a subsection on link prediction, in which we provide an explanation of the relevant GNNs, which is this time preceded by an explanation of the baselines, i.e., traditional link prediction methods including both heuristics and latent-feature methods. This is done because the link prediction baselines used in the research project are rather unknown, unlike the baselines of the other two tasks. Finally, we present a complex GNN for time series forecasting, whose individual components we explain step by step.

3.1 General framework of graph neural networks

Graph neural networks are neural network architectures specifically designed for learning representations of graph-structured data including learning node representations of big graphs, e.g., social networks and the World Wide Web, and learning representations of entire graphs, e.g., molecular graphs. Our general framework focuses on learning node representations for large-scale graphs. A variety of graph neural networks have been proposed (Kipf and Welling (2017); Veličković et al. (2018); Gilmer et al. (2017); Xhonneux et

al. (2020); Liao et al. (2019); Kipf and Welling (2016); Veličković et al. (2019)). We refer to Wu et al. (2022) for this subsection.
Table 1 summarizes the notation used throughout this section.

Concept	Notation
Graph	$\mathcal{G} = (\mathcal{V}, \mathcal{E})$
Adjacency matrix	$A \in \mathbb{R}^{N \times N}$
Node features	$X \in \mathbb{R}^{N \times C}$
Total number of GNN layers	K
Node representations at the k-th layer	$H^k \in \mathbb{R}^{N \times F}, k \in \{1, 2, \ldots, K\}$

Table 1: Notation used throughout this section

The central idea behind graph neural networks is to iteratively update the node representations by combining the representations of their neighbors and their own representations. We now get to know the general framework of graph neural networks of Xu et al. (2019). Given the initial node representation $H^0 = X$, in each layer there are two important functions:

- **AGGREGATE**, which tries to aggregate the information from the neighbors of each node;
- **COMBINE**, which tries to update the node representations by combining the aggregated information form neighbors with the current node representations.

Based on this, we can mathematically define the general framework of graph neural networks as follows:
Initialization: $H^0 = X$
For $k = 1, 2, \ldots, K$,

$$a_v^k = \textbf{AGGREGATE}^k\{H_u^{k-1} : u \in N(v)\} \tag{3.1}$$

$$H_v^k = \textbf{COMBINE}^k\{H_v^{k-1}, a_v^k\}, \tag{3.2}$$

with $N(v)$ being the set of neighbors of the v-th node. Here, the node representations H^k in the last layer are the final node representations. These node representations can then be employed in downstream tasks. In the node classification task, for instance, the label of node v, denoted as $\hat{y}_v$, can be predicted through a softmax function, i.e.,

$$\hat{y}_v = \text{Softmax}(WH_v^T), \tag{3.3}$$

with $W \in \mathbb{R}^{|\mathcal{L}| \times F}$ and $|\mathcal{L}|$ being the number of labels in the output space. With a given set of labeled nodes, the whole model can be trained by minimizing the following loss function:

$$O = \frac{1}{n_l} \sum_{i=1}^{n_l} \text{loss}\left(\hat{y}_i, y_i\right), \tag{3.4}$$

where y_i is the ground truth label of node i, n_l is the number of labeled nodes, $loss\left(\cdot, \cdot\right)$ is a loss function such as cross entropy loss function. The optimization of the entire neural network can be done by minimizing the objective function O with backpropagation.

Having introduced a general framework of graph neural networks, we now proceed to presenting the GNNs for the three learning tasks.

3.2 Node classification

Node classification aims to classify the nodes of a graph into a few predefined categories. For instance, it can be applied to predict the political bias of each user in social networks, to predict the function role of each protein in protein-protein interaction networks, and to classify web pages into different semantic categories in the World Wide Web. Obtaining accurate predictions requires having very effective node representations, which play a key role in the performance of node classification. Such node representations can be learned in the context of the above general framework by combining the graph structure information and the node features.

We now cover a number of widely used graph neural networks for node classification and refer to Wu et al. (2022). In this chapter, we restrict ourselves to semi-supervised approaches and eventually take a look at over-smoothing, a common problem of graph neural networks for node classification.

3.2.1 Graph convolutional networks

Graph convolutional networks (Kipf and Welling (2017)) are the most popular graph neural network architecture due to their simplicity and effectiveness in a variety of tasks and applications. Specifically, the propagation rule updates the node representations in the following way:

$$H^{k+1} = \sigma\left(\tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}} H^k W^k\right). \tag{3.5}$$

Here, $\tilde{A} = A + I$ is the adjacency matrix of the given undirected graph $\mathcal{G}$ with added self-connections, so that node features themselves can be incorporated when updating the node representations. $I \in \mathbb{R}^{N \times N}$ is the identity matrix, $\tilde{D}$ is a diagonal matrix with $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$ and $\sigma(\cdot)$ is an activation function such as $\text{ReLU}(\cdot) = \max(0, \cdot)$. $W^k \in \mathbb{R}^{F \times F'}$ is a layer-specific trainable weight matrix trained during the optimization, where F and F' are the dimensions of node representations in the k-th and $(k+1)$-th layer, respectively.
Equation 3.5 can be broken down in such a way that it becomes clear how the AGGREGATE and COMBINE functions are defined in the GCN. For node i, the node update equation can be rewritten as follows:

$$H_i^k = \sigma\left(\sum_{j \in \{N(i) \cup i\}} \frac{\tilde{A}_{ij}}{\sqrt{\tilde{D}_{ii}\tilde{D}_{jj}}} H_j^{k-1} W^k\right) \tag{3.6}$$

$$H_i^k = \sigma\left(\sum_{j \in N(i)} \frac{A_{ij}}{\sqrt{\tilde{D}_{ii}\tilde{D}_{jj}}} H_j^{k-1} W^k + \frac{1}{\tilde{D}_i} H_i^{k-1} W^k\right). \tag{3.7}$$

Equation 3.7 shows us that the AGGREGATE function is defined as the weighted average of the neighbor node representations, where the weight of the neighbor j is determined by the weight of the edge between i and j, i.e., A_{ij} normalized by the degrees of the two nodes. The COMBINE function is defined as the summation of the aggregated messages and the node representation itself, in which the node representation is normalized by its own degree.

Connections with spectral graph convolutions

Now, we look into the connections between GCNs and traditional spectral filters defined on graphs (Defferrard et al. (2016)). The spectral convolutions on graphs can be defined as a multiplication of a node-wise signal $\mathbf{x} \in \mathbb{R}^N$ with a convolutional filter $g_\theta = \text{diag}(\theta)$ in the Fourier domain, where $\theta \in \mathbb{R}^N$ is the parameter of the filter. Mathematically,

$$g_\theta \star \mathbf{x} = U g_\theta U^T \mathbf{x}, \tag{3.8}$$

where U stands for the matrix of eigenvectors of the normalized graph Laplacian matrix $L = I_N - D^{-\frac{1}{2}} A D^{-\frac{1}{2}}$. Specifically, $L = U \Lambda U^T$, where Λ is a diagonal matrix of eigenvalues, and $U^T \mathbf{x}$ is the graph Fourier transform of the input signal $\mathbf{x}$. We can understand g_θ as a function of the eigenvalues of the normalized graph Laplacian matrix L, i.e., $g_\theta(\Lambda)$. In practice, the direct calculation of equation 3.8 is very computationally expensive, which is quadratic to the number of nodes N. Hammond et al. (2011) show that this problem can

be circumvented by approximating the function g_θ with a truncated expansion of Chebyshev polynomials $T_k(x)$ up to K^{th} order:

$$g_{\theta'}(\Lambda) = \sum_{k=0}^{K} \theta'_k T_k(\tilde{\Lambda}), \tag{3.9}$$

with $\tilde{\Lambda} = \frac{2}{\lambda_{\max}}\Lambda - I$, and $\lambda_{\max}$ being the largest eigenvalue of L. $\theta' \in \mathbb{R}^K$ is the vector of Chebyshev coefficients. $T_k(x)$ are Chebyshev polynomials which are recursively defined as $T_k(x) = 2xT_{k-1}(x) - T_{k-2}(x)$, where $T_0(x) = 1$ and $T_1(x) = x$. The combination of equation 3.8 and equation 3.9 allows to rewrite the convolution of a signal $\mathbf{x}$ with a filter $g_{\theta'}$ as follows:

$$g_{\theta'} \star \mathbf{x} = \sum_{k=0}^{K} \theta'_k T_k(\tilde{L})\mathbf{x}, \tag{3.10}$$

where $\tilde{L} = \frac{2}{\lambda_{\max}}L - I$. It is evident that each node only depends on the information within the K^{th}-order neighborhood. The evaluation of equation 3.10 comes with an overall complexity of $\mathcal{O}(|\mathcal{E}|)$, which is very efficient.

Layer-wise linear model

In order to arrive at a neural network based on graph convolutions, we can stack multiple convolutional layers defined in equation 3.10 with each layer followed by a nonlinear transformation. At each layer, instead of being limited to the explicit parameterization by the Chebyshev polynomials defined in equation 3.10, the authors of the GCN proposed to limit the number of convolutions to $K = 1$ at each layer. This ensures that, at each layer, only a linear function over the graph Laplacian matrix L is defined. By stacking multiple such layers, however, we can still cover a rich class of convolution filter functions on graphs. Intuitively, such a model can alleviate the problem of overfitting local neighborhood structures for graphs whose node degree distribution has a high variance such as social networks, the World Wide Web, and citation networks.
Furthermore, we can approximate $\lambda_{\max}$ at each layer, which could be accommodated by the neural network parameters during training. After all these simplifications, we obtain

$$g_{\theta'} \star \mathbf{x} \approx \theta'_0\mathbf{x} + \theta'_1\mathbf{x}(L - I_N)\mathbf{x} = \theta'_0\mathbf{x} - \theta'_1 D^{-\frac{1}{2}}AD^{-\frac{1}{2}}, \tag{3.11}$$

with θ'_0 and θ'_1 being two free parameters, which could be shared over the entire graph. The number of parameters can be reduced even further, so that overfitting can be reduced

and meanwhile the number of operations per layer can be minimized. This leads us to the following expression:

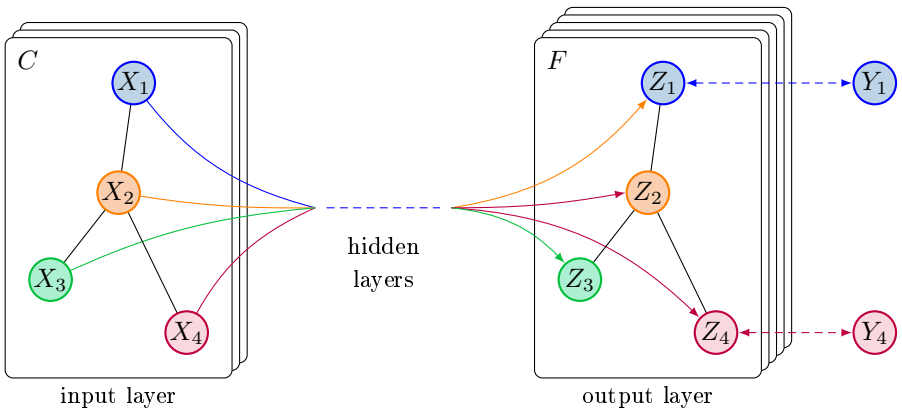
$$g_\theta \star \mathbf{x} \approx \theta(I + D^{-\frac{1}{2}} A D^{-\frac{1}{2}})\mathbf{x}, \tag{3.12}$$

where $\theta = \theta'_0 = -\theta'_1$. The matrix $I + D^{-\frac{1}{2}} A D^{-\frac{1}{2}}$ can cause problems since its eigenvalues lie in the interval of $[0, 2]$ and in a deep graph convolutional neural network, repeated application of the above function will likely cause exploding or vanishing gradients and hence numerical instabilities. To solve this, we further normalize this matrix by converting $I + D^{-\frac{1}{2}} A D^{-\frac{1}{2}}$ to $\tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}}$, where $\tilde{A} = A + I$, and $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$.

So far, we have only considered the case that there is only one feature channel and one filter. This can be easily generalized to an input signal with C channels $X \in \mathbb{R}^{N \times C}$ and F filters, the number of hidden units, as follows:

$$H = \tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}} X W, \tag{3.13}$$

with $W \in \mathbb{R}^{C \times F}$ being a matrix of filter parameters. H is the convolved signal matrix. Figure 2 (Kipf and Welling (2017)) shows this: a multi-layer GCN with C input channels and F feature maps in the output layer.



Graph convolutional network

Schematic illustration of a multi-layer GCN for semi-supervised learning with C input channels and F feature maps in the output layer. The graph structure as indicated by the edges in black is shared over layers and labels are denoted by Y_i.

Figure 2: Multi-layer graph convolutional network for semi-supervised learning

3.2.2 Graph attention networks

For the GCNs presented above, the importance of a neighbor j for a target node i is determined by the weight of their edge A_{ij} normalized by their node degrees. In practice, however, the input graph may be noisy, so that the weights may not represent the true strength between two nodes. For this reason, it may be more sensible to automatically learn the importance of each neighbor. VELIČKOVIĆ ET AL. (2018) incorporate this idea into their model, Graph Attention Network (GAT), the aim of which is to learn the importance of each neighbor based on the attention mechanism (BAHDANAU ET AL. (2015); VASWANI ET AL. (2017)). The attention mechanism has found wide application in tasks such as natural language understanding, e.g., machine learning translation and question answering, and computer vision, e.g., visual question answering and image captioning. In the following, we learn how the attention mechanism comes into play in GNNs.

Graph attention layer

The graph attention layer specifies how to obtain the new node representations at layer k, $H^k \in \mathbb{R}^{N \times F'}$, from the hidden node representations at layer $k-1$, $H^{k-1} \in \mathbb{R}^{N \times F}$. By applying a shared linear transformation $W \in \mathbb{R}^{F \times F'}$ to every node, sufficient expressive power to transform the lower-level node representations to higher-level node representations is guaranteed. Then, self-attention is performed on the nodes, yielding the attention coefficients for any pair of nodes through a shared attentional mechanism $a : \mathbb{R}^{F'} \times \mathbb{R}^{F'} \to \mathbb{R}$

$$e_{ij} = a(WH_i^{k-1}, WH_j^{k-1}). \tag{3.14}$$

Here, e_{ij} indicates the relationship strength between node i and j. Note that in this subsection H_i^{k-1} denotes a column-wise vector instead of a row-wise vector. In principle, it is possible that each node attends to every other node on the graph. This, however, ignores the graph structural information. Therefore, it would be more reasonable for a node to only attend to its neighbors. In practice, the first-order neighbors are only used, including the node itself. And to achieve comparability of coefficients across different nodes, the attention coefficients are usually normalized with the softmax function:

$$\alpha_{ij} = \text{Softmax}_j(\{e_{ij}\}) = \frac{\exp(e_{ij})}{\sum_{l \in N(i)} \exp(e_{il})}. \tag{3.15}$$

It becomes apparent that for a node i, α_{ij} basically defines a multinomial distribution over its neighbors, also interpretable as the transition probability from node i to each of its

neighbors.

In their work, Veličković et al. (2018) define the attention mechanism a as a single-layer feedforward neural network including a linear transformation with the weight vector $W_2 \in \mathbb{R}^{1\times 2F'}$ and a LeakyReLU nonlinear activation function with negative input slope $\alpha = 0.2$. This leads us to the following architecture, which gives us the attention coefficients:

$$\alpha_{ij} = \frac{\exp(\text{LeakyReLU}(W_2[WH_i^{k-1} \| WH_j^{k-1}]))}{\sum_{l\in N(i)} \exp(\text{LeakyReLU}(W_2[WH_i^{k-1} \| WH_l^{k-1}]))}, \tag{3.16}$$

with $\|$ representing the operation of concatenating two vectors. Now, the new node representation is a linear combination of the neighboring node representations with the weights determined by the attention coefficients, which can receive a subsequent nonlinear transformation, i.e.,

$$H_i^k = \sigma\left(\sum_{j\in N(i)} \alpha_{ij} W H_j^{k-1}\right). \tag{3.17}$$

The attention mechanism is illustrated in the left part of Figure 3 (Veličković et al. (2018)).

Multi-head attention

In practical applications, *multi-head attention* can be deployed rather than using only single attention mechanism, where each attention mechanism determines a different similarity function over the nodes. For each attention head, we can independently receive a new node representation according to equation 3.17, the final node representation resulting as a concatenation of the node representations learned by different attention heads. Mathematically, this corresponds to:

$$H_i^k = \Big\|_{t=1}^{T} \sigma\left(\sum_{j\in N(i)} \alpha_{ij}^t W^t H_j^{k-1}\right), \tag{3.18}$$

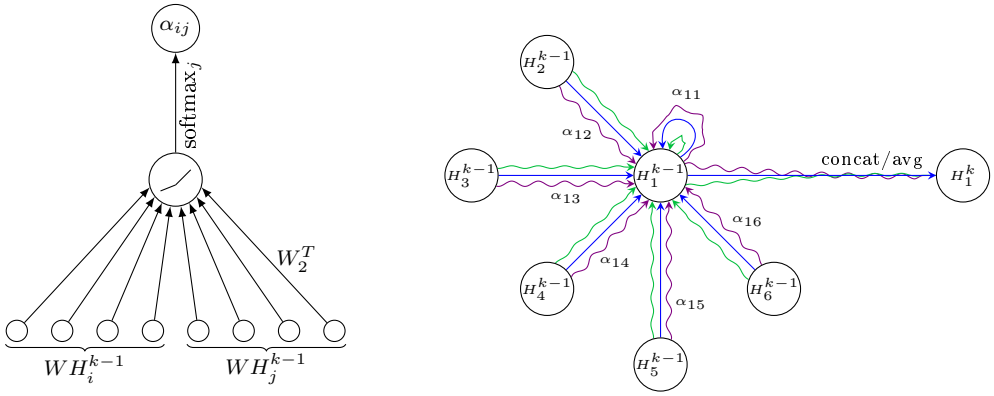
with T being the total number of attention heads, α_{ij}^t being the attention coefficient calculated from the t-th attention head and W^t being the linear transformation matrix of the t-th attention head.

In their work, Veličković et al. (2018) remark that in the final layer, when combining the node representations from different attention heads, instead of using the operation of concatenation, other pooling techniques can be employed, e.g., merely taking the average

node representations from different attention heads:

$$H_i^k = \sigma\left(\frac{1}{T}\sum_{t=1}^{T}\sum_{j\in N(i)}\alpha_{ij}^t W^t H_j^{k-1}\right). \tag{3.19}$$

The aggregation process of a multi-head graph attentional layer is illustrated in the right part of Figure 3 (Veličković et al. (2018)).



The left figure depicts the attention mechanism employed in the model of Veličković et al. (2018), parameterized by a weight vector $W_2 \in \mathbb{R}^{1\times 2F'}$, applying a LeakyReLU activation. The right figure illustrates multi-head attention with $K = 3$ heads by node 1 on its neighborhood. Here, different arrow styles and colors stand for independent attention computations. The aggregated features from each head are concatenated or averaged to obtain H_1^k.

Figure 3: Attention mechanism and multi-head attention

3.2.3 Over-smoothing problem

Stacking multiple layers of GNNs when training deep GNNs normally causes results to deteriorate, a problem observable in many different GNN architectures. The underlying problem is over-smoothing, with Li et al. (2018a) being the first ones to explicitly study this problem. In their work, they show that the GCN (Kipf and Welling (2017)) is a special case of Laplacian smoothing:

$$Y = (1-\gamma I)\,X + \gamma\tilde{A}_{rw}X, \tag{3.20}$$

where $\tilde{A}_{rw} = \tilde{D}^{-1}\tilde{A}$, which defines the transitional probabilities between nodes on graphs. With $\gamma = 1$ and using $\tilde{A}_{\text{sym}} = \tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}$, the GCN is a special case of Laplacian smoothing. Laplacian smoothing is beneficial for downstream tasks such as node classification insofar as

it pushes nodes belonging to the same clusters to take similar representations. In deep GCNs, however, all the nodes have similar representations and therefore suffer from the problem of over-smoothing. Consequently, we get inferior results in downstream tasks. Over-smoothing is also the subject of the work of Zhao and Akoglu (2020), Li et al. (2018a), Xu et al. (2018), Li et al. (2019a) and Rong et al (2020). For PairNorm, a method to mitigate the problem of over-smoothing, refer to Zhao and Akoglu (2020).

3.3 Link prediction

Link prediction is the problem of predicting the existence of a link between two nodes in a graph (Liben-Nowell and Kleinberg (2007)). It finds widespread application in fields such as friend recommendation in social networks (Adamic and Adar (2003)), co-authorship prediction in citation networks (Shibata et al. (2012)), movie recommendation in Netflix (Bennett and Lanning (2007)), protein interaction prediction in biological networks (Qi et al. (2006)), drug response prediction (Stanfield et al. (2017)), metabolic network construction (Oyetunde et al. (2017)), hidden terrorist group identification (Al Hasan and Zaki (2011)) and knowledge graph completion (Nickel et al. (2016)). For this subsection, we refer to Wu et al. (2022).
In the following, we dive into link prediction methods and perform an in-depth review of them. This means that in addition to GNN methods, we will also look at more basic link predictions methods. This is done against the background that the latter are in general relatively unknown, but nevertheless needed as baselines for our research project. Overall, our approach is as follows: We start with traditional link prediction methods, including heuristics, latent-feature methods as well as content-based methods. After that, we turn to the more advanced GNN methods for link prediction, which encompass both node-based methods and subgraph-based methods.

3.3.1 Traditional link prediction methods

There are mainly three types of traditional link prediction methods: heuristic methods, latent-feature methods, and content-based methods.

3.3.1.1 Heuristic methods

Heuristic methods make use of simple but effective node similarity scores to predict the likelihood of a link (Liben-Nowell and Kleinberg (2007); Lü and Zhou (2011)). In the following, x and y are used to denote the source and target node between which to predict

a link. Here, $\Gamma(x)$ denotes the set of x's neighbors.

Local heuristics

The simplest heuristic is called *common neighbors* (CN) and counts the number of neighbors that two nodes share as a measurement of their likelihood of having a link:

$$f_{\mathrm{CN}}(x, y) = |\Gamma(x) \cap \Gamma(y)|. \tag{3.21}$$

CN comes into play in social network friend recommendation. Its underlying assumption is that the more common friends two people have, the more likely it is that they themselves are also friends.

Jaccard score (JC), on the other hand, measures the proportion of common neighbors:

$$f_{\mathrm{JC}}(x, y) = \frac{|\Gamma(x) \cap \Gamma(y)|}{|\Gamma(x) \cup \Gamma(y)|}. \tag{3.22}$$

The *preferential attachment* (PA) heuristic (Barabási and Albert (1999)) uses the product of node degrees to measure the link likelihood:

$$f_{\mathrm{PA}}(x, y) = |\Gamma(x)| \cdot |\Gamma(y)|. \tag{3.23}$$

PA assumes that x is more likely to be connected to y if y has a high degree. For instance, in citation networks, a new paper is more likely to cite those papers which already have a lot of citations. Networks formed by the PA mechanism are called scale-free networks (Barabási and Albert (1999)).

The three heuristics presented above only involve one-hop neighbors of two target nodes. Therefore, they are called *first-order heuristics*. Next, two *second-order heuristics* are introduced.

The *Adamic-Adar* (AA) heuristic (Adamic and Adar (2003)) considers the weight of common neighbors:

$$f_{\mathrm{AA}}(x, y) = \sum_{z \in \Gamma(x) \cap \Gamma(y)} \frac{1}{\log|\Gamma(z)|}, \tag{3.24}$$

where a high-degree common neighbor z is weighted less due to being downweighted by the reciprocal of $\log|\Gamma(z)|$. Here, the assumption is that a high-degree node connecting to both x and y is less informative than a low-degree node.

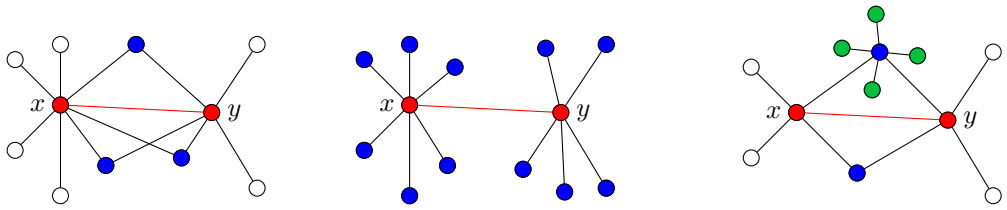
Resource allocation (RA) (Zhou et al. (2009)) uses a more aggressive downweighting

factor:

$$f_{\mathrm{RA}}(x,y) = \sum_{z \in \Gamma(x) \cap \Gamma(y)} \frac{1}{|\Gamma(z)|}, \tag{3.25}$$

hence, it favors low-degree common neighbors more.

Both AA and RA are second-order heuristics because up to two hops of neighbors of x and y are required to compute the score. Both first-order and second-order heuristics are local heuristics, as they can all be computed from a local subgraph around the target link without the need to know the entire graph. Three local heuristics, CN, PA, and AA, are illustrated in Figure 4 (Wu et al. (2022)).



(a) common neighbors (CN): $|\Gamma(x) \cap \Gamma(y)|$ (b) preferential attachment (PA): $|\Gamma(x)| \cdot |\Gamma(y)|$ (c) Adamic-Adar (AA): $\sum_{z \in \Gamma(x) \cap \Gamma(y)} \frac{1}{\log |\Gamma(z)|}$

Figure 4: Illustration of three link prediction heuristics

Global heuristics

There are also *high-order heuristics* which require knowing the entire network. Three of them are given by Katz index (KA) (Katz (1953)), rooted PageRank (RPR) (Brin and Page (2012)), and SimRank (SR) (Jeh and Widom (2002)).

Katz index uses a weighted sum of all the walks between x and y where a longer walk is discounted more:

$$f_{\mathrm{KA}}(x,y) = \sum_{l=1}^{\infty} \beta^l |\mathrm{walks}^{(l)}(x,y)|, \tag{3.26}$$

where β is a decaying factor between 0 and 1, and $|\mathrm{walks}^{(l)}(x,y)|$ counts the length-l walks between x and y. If only length-2 are considered, Katz index reduces to CN.

Rooted PageRank (RPR) is a generalization of PageRank. It first computes the stationary distribution π_x of a random walker starting from x who randomly moves to one of its current neighbors with probability α, or returns to x with probability $1-\alpha$. Then it uses π_x at node y (denoted by $[\pi_x]_y$) to predict link (x,y). In the case of an undirected graph, a symmetric

version of rooted PageRank uses

$$f_{\mathrm{RPR}}(x,y) = [\pi_x]_y + [\pi_y]_x, \tag{3.27}$$

to predict the link.

The **SimRank** (SR) score assumes that two nodes are similar if their neighbors are also similar. It is defined recursively: if $x = y$, then $f_{\mathrm{SR}}(x,y) := 1$; otherwise,

$$f_{\mathrm{SR}}(x,y) = \gamma \frac{\sum_{a\in\Gamma(x)} \sum_{b\in\Gamma(y)} f_{\mathrm{SR}}(a,b)}{|\Gamma(x)| \cdot |\Gamma(y)|}, \tag{3.28}$$

where γ is a constant between 0 and 1.

High-order heuristics are global heuristics. By computing node similarity from the entire graph, high-order heuristics often have better performance than first-order and second-order heuristics.

Overall, heuristic methods can be regarded as computing predefined graph structure features located in the observed node and edge structures of the graph. Despite being effective in many domains, these handcrafted graph structure features have limited expressivity - they only capture a small subset of all possible structure patterns, and cannot express general graph structure features underlying different graphs. Moreover, heuristic methods only work well when the graph formation mechanism aligns with the heuristic. There may be networks with complex formation mechanisms which no existing heuristic can capture well. Most heuristics only work for homogeneous graphs.

3.3.1.2 Latent-feature methods

Latent-feature methods are the second class of traditional link prediction methods and also go by the names of latent-factor models and embedding methods in some literature. They compute latent properties or representation of nodes, often obtained by factorizing a specific matrix derived from the network, such as the adjacency matrix and the Laplacian matrix. These latent features of nodes are not directly observable - they must be computed from the graph through optimization. Latent features are also not interpretable. This means that we do not know what each latent feature dimension describes, unlike explicit node features where each feature dimension represents a specific property of nodes.

Matrix factorization

Matrix factorization (Koren et al. (2009); Ahmed et al. (2013)) is a popular method

that originated from the recommender systems literature. In matrix factorization, the observed adjacency matrix A of the network is factorized into the product of a low-rank latent embedding matrix Z and its transpose. This means that matrix factorization approximately reconstructs the edge between i and j using their k-dimensional latent embeddings $\mathbf{z}_i$ and $\mathbf{z}_j$:

$$\hat{A}_{i,j} = \mathbf{z}_i^T \mathbf{z}_j. \tag{3.29}$$

It then minimizes the mean squared error between the reconstructed adjacency matrix and the true adjacency matrix over the observed edges to learn the latent embeddings:

$$\mathcal{L} = \frac{1}{|\mathcal{E}|} \sum_{(i,j)\in\mathcal{E}} \left(A_{i,j} - \hat{A}_{i,j}\right)^2. \tag{3.30}$$

Finally, new links can be predicted by the inner product between nodes' latent embeddings. Variants of matrix factorization include using powers of A (Cangea et al. (2018)) and using general node similarity matrices (Ou et al. (2016)) to replace the original adjacency matrix A. If we replace A with the Laplacian matrix L and define the loss as follows:

$$\mathcal{L} = \sum_{(i,j)\in\mathcal{E}} \|\mathbf{z}_i - \mathbf{z}_j\|_2^2, \tag{3.31}$$

then the nontrivial solution to the above is constructed by the eigenvectors corresponding to the k smallest nonzero eigenvalues of L, which recovers the Laplacian eigenmap technique (Belkin and Niyogi (2001)) and the solution to spectral clustering (von Luxburg (2007)).

Network embedding

Network embedding techniques were influenced by the pioneering work DeepWalk (Perozzi et al. (2014)). They learn low-dimensional representations, so-called embeddings, for nodes, often based on training a skip-gram model (Mikolov et al. (2013)) over random-walk-generated node sequences, so that nodes which often appear nearby each other in a random walk, i.e., nodes close in a graph, have similar representations. Then, the pairwise node embeddings are aggregated as link representations for link prediction. Although network embedding techniques do not explicitly factorize a matrix, it is shown in Qiu et al. (2018) that they implicitly factorize some matrix representations of the graph. Therefore, network embedding techniques can also be categorized into latent-feature methods.
For instance, DeepWalk approximately factorizes:

$$\log\left(\operatorname{vol}(\mathcal{G})\left(\frac{1}{w}\sum_{r=1}^{w}\left(D^{-1}A\right)^{r}\right)D^{-1}\right)-\log(b), \tag{3.32}$$

where $\operatorname{vol}(\mathcal{G})$ is the sum of node degrees, D is the diagonal matrix, w is skip-gram's window size, and b is a constant. From this it is evident that DeepWalk factorizes the log of some high-order normalized adjacency matrices' sum up to w. We can think of a random walk as extending a node's neighborhood to w hops away, so that we not only require direct neighbors to have similar embeddings, but also require nodes reachable from each other through w steps to have similar embeddings.

In a similar way, the LINE algorithm (Tang et al. (2015)) in its second-order forms implicitly factorizes:

$$\log\left(\operatorname{vol}(\mathcal{G})\left(D^{-1}AD^{-1}\right)\right)-\log(b). \tag{3.33}$$

Another population network embedding method is node2vec. It extends DeepWalk with negative sampling and biased random walk and also implicitly factorizes a matrix. The matrix does not have a closed form due to the use of second-order biased random walks (Qiu et al. (2018)).

Latent-feature methods can take global properties and long-range effects into node representations, because all node pairs are used together to optimize a single objective function, and the final embedding learned for a node can be influenced by all nodes in the same connected component during the optimization. However, latent-feature methods cannot capture structural similarities between nodes (Ribeiro et al. (2017)), i.e., two nodes sharing identical neighborhood structures are not mapped to similar embeddings. Latent-feature methods also need an extremely large dimension to express some simple heuristics (Nickel et al. (2014)), so that they sometimes have worse performance than heuristic methods. Finally, latent-feature methods are transductive learning methods: The learned node embeddings cannot generalize to new nodes or new graphs.

3.3.1.3 Content-based methods

Heuristics methods and latent-feature methods face the cold-start problem. When a new node joins the graph, heuristic methods and latent-feature methods may not be able to predict its links accurately because it has no or only a few existing links with other nodes. This is where content-based methods can help by leveraging explicit content features associated with nodes for link prediction, which have wide applications in recommender systems

(Lops et al. (2011)). For example, in citation networks, word distributions can be used as content features for papers. In social networks, a user's profile, such as their demographic information and interests, can be used as their content features, where their friendship information, however, belongs to graph structure features because it is calculated from the graph structure. Yet, content-based methods usually have worse performance than heuristic and latent-feature methods due to not leveraging the graph structure. Thus, they are usually used together with the other two types of methods (Koren (2008); Rendle (2010); Zhao et al. (2017)) to enhance link prediction performance.

3.3.2 GNN methods for link prediction

GNN methods combine graph structure features and content features by learning them together in a unified way, exploiting the graph representation learning ability of GNNs.
There are mainly two GNN-based link prediction paradigms, node-based and subgraph-based. Node-based methods aggregate the pairwise node representations learned by a GNN as the link representation. Subgraph-based methods extract a local subgraph around each link and use the subgraph representation learned by a GNN as the link representation.

3.3.2.1 Node-based methods

The most straightforward way of using GNNs for link prediction is to treat GNNs as inductive network embedding methods which learn node embeddings from local neighborhood, and then aggregate the pairwise node embeddings of GNNs to construct link representations.

Graph autoencoder

Graph Autoencoder (GAE) (Kipf and Welling (2016)) is the pioneering work of node-based methods. Given the adjacency matrix A and node feature matrix X of a graph, GAE (Kipf and Welling (2016)) first uses a GCN (Kipf and Welling (2017)) to compute a node representation $\mathbf{z}_i$ for each node i, and then uses $\sigma\left(\mathbf{z}_i^T \mathbf{z}_j\right)$ to predict link (i, j):

$$\hat{A}_{i,j} = \sigma\left(\mathbf{z}_i^T \mathbf{z}_j\right), \text{ where } \mathbf{z}_i = Z_{i,:}, Z = \mathrm{GCN}\left(X, A\right). \tag{3.34}$$

Z is the node representation, i.e., embedding, matrix output by the GCN, with the i^{th} row of Z being node i's representation $\mathbf{z}_i$. $\hat{A}_{i,j}$ is the predicted probability for link (i, j) and σ is the sigmoid function. If X is not given, GAE can use the one-hot encoding matrix I instead. The model is trained to minimize the cross entropy between the reconstructed adjacency

matrix and the true adjacency matrix:

$$\mathcal{L} = \sum_{i \in \mathcal{V}, j \in \mathcal{V}} \left(-A_{i,j} \log\left(\hat{A}_{i,j}\right) - \left(1 - A_{i,j}\right) \log\left(1 - \hat{A}_{i,j}\right) \right). \tag{3.35}$$

In practice, the loss of positive edges $\left(A_{i,j} = 1\right)$ is upweighted by k, where k is the ratio between negative edges $\left(A_{i,j} = 0\right)$ and positive edges. The purpose is to balance the positive and negative edges' contribution to the loss. Otherwise, the loss might be dominated by negative edges since a lot of graphs in practice are sparse.

Variational graph autoencoder

The variational version of GAE is called VGAE, or Variational Graph Autoencoder (Kipf and Welling (2016)). Instead of learning deterministic node embeddings $\mathbf{z}_i$, VGAE uses two GCNs to learn the mean μ_i and variance σ_i^2 of $\mathbf{z}_i$, respectively.
VGAE assumes the adjacency matrix A is generated from the latent node embeddings Z through $p\left(A|Z\right)$, where Z follows a prior distribution $p\left(Z\right)$. Similar to GAE, VGAE uses an inner-product-based link reconstruction model as $p\left(A|Z\right)$:

$$p\left(A|Z\right) = \prod_{i \in \mathcal{V}} \prod_{j \in \mathcal{V}} p\left(A_{i,j}|\mathbf{z}_i, \mathbf{z}_j\right), \text{ where } p\left(A_{i,j} = 1|\mathbf{z}_i, \mathbf{z}_j\right) = \sigma\left(\mathbf{z}_i^T \mathbf{z}_j\right). \tag{3.36}$$

The prior distribution $p\left(Z\right)$ takes a standard Normal distribution:

$$p\left(Z\right) = \prod_{i \in \mathcal{V}} p\left(\mathbf{z}_i\right) = \prod_{i \in \mathcal{V}} \mathcal{N}\left(\mathbf{z}_i|0, I\right). \tag{3.37}$$

With $p\left(A|Z\right)$ and $p\left(Z\right)$ being given, we can compute the posterior distribution of Z using Bayes' rule. Often, however, this distribution is intractable. For this reason, given the adjacency matrix A and node feature matrix X, VGAE uses GNNs to approximate the posterior distribution of the node embedding matrix Z:

$$q\left(Z|X, A\right) = \prod_{i \in \mathcal{V}} q\left(\mathbf{z}_i|X, A\right), \text{ where } q\left(\mathbf{z}_i|X, A\right) = \mathcal{N}(\mathbf{z}_i|\mu_i, \text{diag}(\sigma_i^2)). \tag{3.38}$$

The mean μ_i and variance σ_i^2 of $\mathbf{z}_i$ are obtained by using two GCNs. Then, VGAE maximizes the evidence lower bound to learn the GCN parameters:

$$\mathcal{L} = \mathbb{E}_{q\left(Z|X,A\right)} \left[\log\left(p\left(A|Z\right)\right) \right] - \text{KL}\left[q\left(Z|X, A\right) \parallel p\left(Z\right) \right], \tag{3.39}$$

where $\mathrm{KL}\left[q\left(Z|X,A\right) \parallel p\left(Z\right)\right]$ is the Kullback-Leibler divergence between the approximated posterior and the prior distribution of Z. We optimize the evidence lower bound using the reparametrization trick (Kingma and Welling (2014)). At the end, the embedding means μ_i and μ_j are used to predict link (i,j) by $\hat{A}_{i,j} = \sigma\left(\mu_i^T \mu_j\right)$.

There is a range of variants and extensions of GAE and VGAE. Refer to Wu et al. (2022) for further details.

3.3.2.2 Subgraph-based methods

Subgraph-based methods extract a local subgraph around each target link and learn a subgraph representation through a GNN for link prediction.

The SEAL framework

SEAL (Zhang and Chen (2018)) is the pioneering work of subgraph-based methods. It first extracts an *enclosing subgraph* for each target link to predict, and then applies a graph-level GNN with pooling to classify whether the subgraph corresponds to link existence. The enclosing subgraph around a node set is defined as follows.

Definition (Enclosing subgraph) For a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, given a set of nodes $S \subseteq \mathcal{V}$, the h-hop enclosing subgraph for S is the subgraph $\mathcal{G}_S^h$ induced from $\mathcal{G}$ by the set of nodes $\bigcup_{j \in S}\{i | d\left(i,j\right) \leq h\}$, where $d\left(i,j\right)$ is the shortest path distance between nodes i and j.

This means that the h-hop enclosing subgraph around a node set S contains nodes within h hops of any node in S, as well as all the edges between these nodes. In some literature, it is also called h-hop local subgraph, h-hop rooted subgraph, or h-hop ego network. In link prediction tasks, the node set S denotes the two nodes between which to predict a link. For instance, when predicting the link between x and y, $S = \{x, y\}$ and $\mathcal{G}_{x,y}^h$ denotes the h-hop enclosing subgraph for link (x,y).
The reason for extracting an enclosing subgraph for each link should be that SEAL aims to automatically learn graph structure features from the network. Bearing in mind that all first-order heuristics can be calculated from the 1-hop enclosing subgraph around the target link and all second-order heuristics can be computed from the 2-hop enclosing subgraph around the target link, SEAL aims to use a GNN to learn general graph structure features,

i.e., supervised heuristics, from the extracted h-hop enclosing subgraphs instead of using predefined heuristics.

Having extracted the enclosing subgraph $\mathcal{G}_{x,y}^{h}$, the next step is node labeling. SEAL uses a Double Radius Node Labeling (DRNL) to assign an integer label to each node in the subgraph as its additional feature. These different labels are meant to be used to differentiate nodes of different roles in the enclosing subgraph. For example, the center nodes x and y are the target nodes between which the target link is located, so they are different from the rest nodes and should be distinguished. Similarly, nodes at different hops with respect to x and y may have different structural importance to link existence and can therefore also be assigned different labels. A proper node labeling such as DRNL is vital to the success of subgraph-based link prediction methods. Subgraph-based methods have a better link representation learning ability than node-based methods.

DRNL works as follows: First, give label 1 to x and y. Then, for any node i with radius $\big(d\left(i,x\right),d\left(i,y\right)\big)=(1,1)$, assign label 2. Nodes with radius $(1,2)$ or $(2,1)$ get label 3. Nodes with radius $(1,3)$ or $(3,1)$ obtain 4. Nodes with $(2,2)$ get 5. Nodes with $(1,4)$ or $(4,1)$ are assigned 6. Nodes with $(2,3)$ or $(3,2)$ get 7. This proceeds so on and so forth. We see that DRNL iteratively assigns larger labels to nodes with larger radius with respect to the two center nodes.

DRNL satisfies two important criteria:

1. The two target nodes x ad y always get the distinct label 1 so that they can be distinguished from the context nodes.

2. Nodes i and j get the same label if and only if their double radius is the same, i.e., i and j have the same distances to (x,y). So, nodes of the same relative positions within the subgraph described by the double radius $\big(d\left(i,x\right),d\left(i,y\right)\big)$ always have the same label.

There is also a closed-form solution of DRNL for directly mapping $\big(d\left(i,x\right),d\left(i,y\right)\big)$ to labels:

$$l\left(i\right)=1+\min\left(d_x,d_y\right)+\frac{d}{2}\left(\frac{d}{2}+d\%2-1\right), \tag{3.40}$$

with $d_x:=d\left(i,x\right)$, $d_y:=d\left(i,y\right)$, $d:=d_x+d_y$, $\frac{d}{2}$ and $d\%2$ being the integer quotient and remainder of d divided by 2, respectively. For nodes with $d\left(i,x\right)=\infty$ or $d\left(i,y\right)=\infty$, DRNL assigns them a null label 0.

After obtaining the DRNL labels, SEAL transforms them into one-hot encoding vectors, or feeds them to an embedding layer to get their embeddings. These new feature vectors are

concatenated with the original node content features, should there be any, to form the new node features. Yet, its experimental results show that adding pretrained node embeddings does not show clear benefits to the final performance (Zhang and Chen (2018)). In addition, adding pretrained node embeddings even makes SEAL lose the inductive learning ability.

At the end, SEAL feeds these enclosing subgraphs along with their new node feature vectors into a graph-level GNN, DGCNN (Zhang et al. (2018)), in order to learn a graph classification function. Here, the ground truth of each subgraph is whether the two center nodes really have a link. To train this GNN, SEAL randomly samples N existing links from the network as positive training links, and also samples an equal number of unobserved links, i.e., random node pairs, as negative training links. After training, SEAL applies the trained GNN to new unobserved node pairs' enclosing subgraphs to predict their links. The entire SEAL framework is visualized in Figure 5 (Zhang and Chen (2018)).

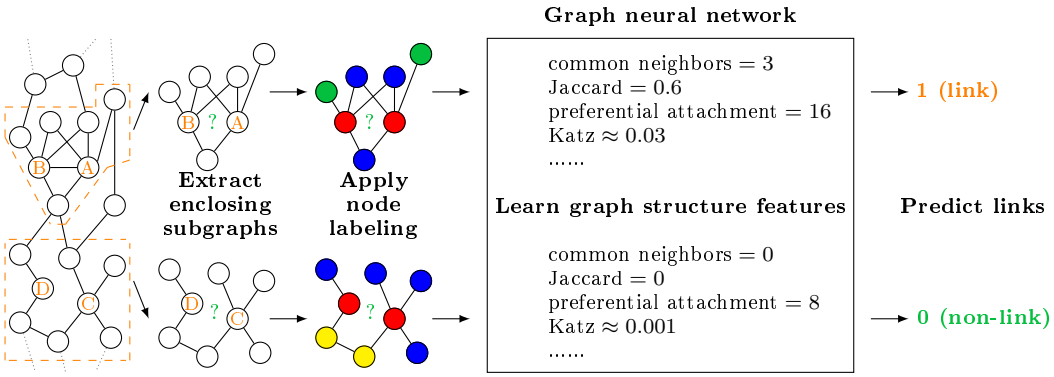


Figure 5: Illustration of the SEAL framework

SEAL demonstrates strong performance for link prediction, achieving consistently superior performance compared to predefined heuristics (Zhang and Chen (2018)).

3.4 Time series forecasting

Time series forecasting is a widely researched field that deals with the prediction of future values based on observed past values. Its applications range from climate studies, market analysis[8] and traffic control to energy grid management (Makridakis et al. (1997)). Many prediction models have been developed that can capture the temporal dynamics of the underlying system. Among them are early autoregressive approaches (Hamilton (1994);

[8]Our application has such a focus, as we predict sales.

Asteriou and Hall (2011)) and recent deep learning methods (Seo et al. (2016); Li et al. (2018b); Yu et al. (2018); Zhao et al. (2019)).

The analysis of univariate time series has been extended to multivariate time series and multiple univariate or multivariate time series. In particular, multivariate predictive models show strong predictive power because they can capture the interdependencies and even causal relationships among variables. An example of this is the vector autoregressive model (Hamilton (1994)), where the coefficient magnitude gives an indication of Granger causality (Granger (1969)) of one variable on another.

Below, we focus on multivariate time series forecasting, i.e., the simultaneous forecasting of multiple time series. In the context of a review of GNNs for time series forecasting, we look at two GNNs to learn how graph structure learning can be incorporated into the optimization framework, and then give an overview of the GNNs that have recently been developed for multivariate time series forecasting. On this basis, we finally consider one of these GNNs in detail. This is a GNN that also addresses graph structure learning and is applied in our research project. In this subsection, we refer to Shang et al. (2021).

3.4.1 Review of GNNs for time series forecasting

In the case of multiple time series, exploratory analysis has shown that pairwise similarities or connections improve forecasting accuracy (Yu et al. (2018)). A well-known example is the traffic network, where each node represents a time series recorded by a particular sensor. The spatial connections of the roads allow us to understand how traffic dynamics propagates along the network. A number of recently developed graph neural network approaches (Seo et al. (2016); Li et al. (2018b); Yu et al. (2018); Zhao et al. (2019)) have the objective to take advantage of this graph structure in the simultaneous forecasting of all time series.

With respect to the graph structure, the problem could arise that it is unavailable or incomplete. Various factors could be responsible for this, e.g., difficult access to such information or deliberate withholding to protect sensitive information. Such circumstances motivate the automatic learning of the hidden graph structure together with the forecasting model.

In the forecasting of multiple interrelated time series, GNN approaches show potential. An example is the work of Franceschi et al. (2019), named LDS, a meta-learning approach in which the graph is used as a hyperparameter in a bilevel optimization framework (Franceschi et al. (2017)). Here, let X_{train} and X_{val} denote the training and the validation sets of time series respectively, $A \in \{0,1\}^{n \times n}$ denote the graph adjacency matrix of the

n series, w denote the parameters used in the GNN, and L and F denote the loss functions used during training and validation, respectively, which may not be identical. In LDS, the problem is cast as learning the probability matrix $\theta \in [0,1]^{n \times n}$, which parametrizes the element-wise Bernoulli distribution from which the adjacency matrix A is sampled:

$$\begin{aligned} \min_{\theta} \quad & E_{A \sim Ber(\theta)} \left[F\left(A, w\left(\theta\right), X_{\text{val}}\right) \right], \\ \text{s.t.} \quad & w\left(\theta\right) = \arg\min_{w} \; E_{A \sim Ber(\theta)} \left[L\left(A, w, X_{\text{train}}\right) \right]. \end{aligned} \tag{3.41}$$

Formulation (3.41) represents a bilevel optimization problem. In the constraint, which by itself is an optimization problem, the GNN weights are defined as a function of the given graph, so that the objective is to optimize such a graph only. For reasons of differentiability, we do not directly operate on the discrete graph adjacency matrix A, but on the continuous probabilities θ instead.

LDS has two disadvantages.

1. Its computation is expensive. The calculation of the derivative of w with respect to θ is obtained by applying the chain rule on a recursive-dynamics surrogate of the inner optimization argmin. Applying the chain rule on this surrogate is equivalent to differentiating an RNN, which is either memory intensive if done in the reverse mode or time consuming when done in the forward mode, when unrolling a deep dynamics.

2. It is challenging to scale. In matrix θ, a total of $\Theta\left(n^2\right)$ entries must be optimized and therefore the method is hard to scale to an increasing number of time series.

To mitigate these disadvantages, Shang et al. (2021) propose the following unilevel optimization:

$$\min_{w} \quad E_{A \sim Ber\left(\theta(w)\right)} \left[F\left(A, w, X_{\text{train}}\right) \right]. \tag{3.42}$$

Formulation (3.42) trains the GNN model in the usual way, the exception being that the probabilities θ - which parameterize the distribution from which A is sampled - are themselves parameterized. These parameters are included, together with the GNN parameters, in the notation w. Although we still use a validation set X_{val} for usual hyperparameter tuning, these hyperparameters are not θ as treated by (3.41). Actually, a second validation set to tune other hyperparameters may be necessary in formulation (3.41).

The approach of Shang et al. (2021) differs from LDS mainly by its parameterization $\theta\left(w\right)$, as opposed to an inner optimization $w\left(\theta\right)$. With the approach of Shang et al. (2021), the modeler can freely design the parameterization and better control the number

of parameters as n^2 increases. Time series representation learning and link prediction techniques provide ample inspiration for modeling. In contrast, LDS is more agnostic since no modeling is required. The effort here lies rather in the nontrivial treatment of the inner optimization.
This means that the approach of Shang et al. (2021) is advantageous in two respects.

1. Its computation is less expensive given that the gradient computation of a unilevel optimization is straightforward and efficient.

2. It better scales given that the number of parameters does not grow quadratically with the number of time series.

Shang et al. (2021) give their method the name *GTS*, short for "*graph for time series*", and thus emphasize the usefulness of graph structure learning for improving time series forecasting. However, it should be noted that the actual purpose of the graph is to enhance the forecasting quality, rather than revealing causal relationships of the series or recovering the ground truth graph. The graph - whether learned or pre-existing - acts as additional information that allows the model to better capture global signals and apply them on each series. In the case of learned graphs, only forecasting accuracy can serve as a golden measure for quality. For example, the traffic network does not necessarily provide the best pairwise relationships that a GNN can use to forecast time series. Nevertheless, we include regularization in our model that penalizes significant deviation from our prior belief, and thus can make our GTS more robust. If a certain ground truth graph is assumed, then the learned graph will be a related version of it for a more accurate forecast.

Related work

Early textbook methods are autoregressive and vector autoregressive models (Hamilton (1994)), autoregressive integrated moving average (ARIMA) (Asteriou and Hall (2011)), hidden Markov models (HMM) (Baum and Petrie (1966)), and Kalman filters (Zarchan and Musoff (2000)). What these methods have in common is that they are linear models that use a window of past information to predict the next time step. However, nonlinear versions with parameterization were also proposed.
RNN (Williams et al. (1986)) is a nonlinear extension and later evolved into LSTM (Hochreiter and Schmidhuber (1997)), BiLSTM (Schuster and Paliwal (1997)), and GRU (Cho et al. (2014)), which addresses several limitations of the vanilla RNN,

such as the vanishing gradient problem. However, these architectures are hard to parallelize due to the recurrent nature of the forward and backward computation. Therefore, architectures such as Transformer (Vaswani et al. (2017)) and BERT (Devlin et al. (2019)) have recently been developed to address parallelization by introducing attention mechanisms that simultaneously digest past and future information. Even though these models are more widely used for sequence data under the context of natural language processing, they are just as directly useful for time series (Shih et al. (2019); Li et al. (2019b)).

Graph neural networks have been developed in deep learning to handle graph-structured data. Here, graph nodes are represented by feature vectors. For time series, however, several specialized architectures have recently been proposed: e.g., GCRN (Seo et al. (2016)), DCRNN (Li et al. (2018)), STGCN (Yu et al. (2018)), and T-GCN (Zhao et al. (2019)). These architectures combine the temporal recurrent processing with graph convolution and thus augment the representation learning of the individual time series.

As graph structure learning, not necessarily for time series, appears in different contexts, methods cover a wide range. One field of study is probabilistic graphical models and causal inference, where the directed acyclic structure is enforced. Among the gradient-based approaches in this context are NOTEARS (Zheng et al. (2018)), DAG-GNN (Yu et al. (2019)), and GraN-DAG (Lachapelle et al. (2020)). It should be noted that a general graph can still be useful without resorting to causality. LDS (Franceschi et al. (2019)) is a meta-learning approach that demonstrates to improve the performance on node classification tasks. MTGNN (Wu et al. (2020)) parameterizes the graph as a degree-k graph, which is learned end-to-end with a GNN for forecasting time series. Shang et al. (2021), on the other hand, propose a more general structural prior for the graph. NRI (Kipf et al. (2018)) uses a latent-variable approach and learns a latent graph for forecasting system dynamics.

3.4.2 Graph for time series

In order to fully present the GTS model, we first provide the required notation and then move on to delineate the graph structure parameterization, the forecasting model and the training loss, which can be complemented by an a priori graph.

We first present the notation. Let X denote the training data, a three-dimensional tensor, where the three dimensions are feature, time, and the n series. Superscripts refer to the series and subscripts refer to time; that is, X^i denotes the i-th series for all features and

time and X_t denotes the t-th time step for all features and series. Training involves a total of S time steps. The model will use a window of T steps to forecast the next τ steps. For each valid t, let $\hat{X}_{t+T+1:t+T+\tau} = f\left(A, w, X_{t+1:t+T}\right)$ denote the model, which generates the forecasts $\hat{X}_{t+T+1:t+T+\tau}$ from the observations $X_{t+1:t+T}$ by exploiting the graph structure A and being parameterized by w. We use ℓ to denote the loss function between the prediction and the ground truth so that the typical training objective is given by

$$\sum_t \ell\left(f\left(A, w, X_{t+1:t+T}\right), X_{t+T+1:t+T+\tau}\right). \tag{3.43}$$

The following part deals with the three remaining details: the parameterization A, the model f, and the loss ℓ.

Graph structure parameterization

Since the matrix $A \in \{0,1\}^{n \times n}$ requires a differentiable function that outputs discrete values 0 or 1, it is hard to parameterize. Here, the idea is to let A be a random variable of the matrix Bernoulli distribution parameterized by $\theta \in [0,1]^{n \times n}$, so that A_{ij} is independent for all the (i,j) pairs with $A_{ij} \sim \text{Ber}\left(\theta_{ij}\right)$. The parameter θ_{ij} is the success probability of a Bernoulli distribution. This changes the training objective (3.43) to

$$E_{A \sim \text{Ber}(\theta)}\left[\sum_t \ell\left(f\left(A, w, X_{t+1:t+T}\right), X_{t+T+1:t+T+\tau}\right)\right]. \tag{3.44}$$

As indicated before, θ is parameterized as $\theta\left(w\right)$, because otherwise the n^2 degrees of freedom in θ render the optimization hard to scale. But such a parameterization brings with it a challenge on differentiability if the expectation (3.44) is evaluated through sample average: The gradient of (3.44) does not flow through A in a usual Bernoulli sampling. For this reason, we resort to the Gumbel reparameterization trick proposed by Jang et al. (2017) and Maddison et al. (2017): $A_{ij} = \text{sigmoid}\left(\frac{\log\left(\frac{\theta_{ij}}{1-\theta_{ij}}\right) + \left(g_{ij}^1 - g_{ij}^2\right)}{s}\right)$, with $g_{ij}^1, g_{ij}^2 \sim \text{Gumbel}\left(0,1\right)$ for all i,j. When the temperature $s \to 0$, it holds that $A_{ij} = 1$ with probability θ_{ij} and 0 with remaining probability. In practice, s is progressively annealed in training so that it tends to zero.

In order to parameterize θ, a feature extractor is deployed to yield a feature vector for each series and a link predictor that takes a pair of feature vectors as input and outputs a link probability. For each i, the feature extractor maps a matrix X^i to a vector z^i. Many sequence architectures are applicable; we opt for a simple one. In concrete terms, we perform

convolution along the temporal dimension, vectorize along this dimension, and apply a fully connected layer to reduce the dimension; that is, $z^i = \text{FC}(\text{vec}(\text{Conv}(X^i)))$. The feature extractor is applied on the entire sequence rather than a window of T time steps. Weights are shared among all series.

The link predictor maps pair of vectors $\left(z^i, z^j\right)$ to a scalar $\theta_{ij} \in [0, 1]$. In order to achieve this, we concatenate the two vectors and apply two fully connected layers; that is, $\theta_{ij} = \text{FC}(\text{FC}(z^i||z^j))$. Here, the last activation must be a sigmoid. For the parts of GTS described above, see the top part of Figure 6 (Shang et al. (2021)).

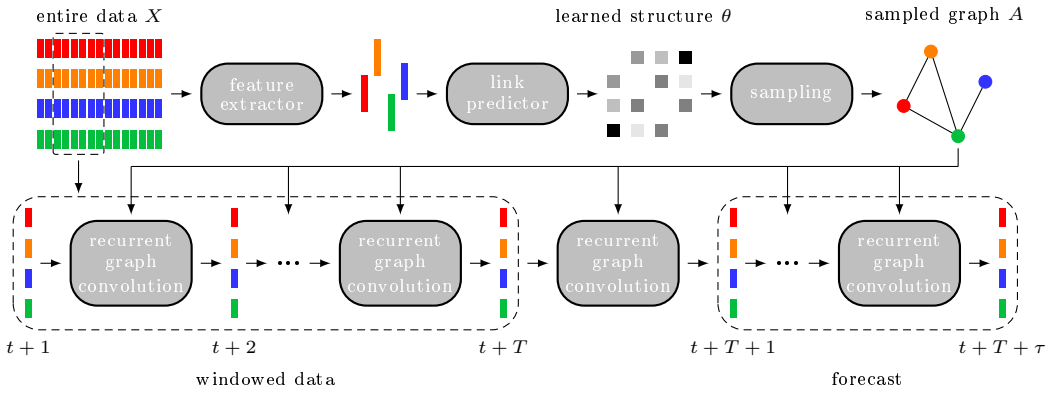


Figure 6: GTS architecture

Graph neural network forecasting

The bottom part of Figure 6 (Shang et al. (2021)) is the forecasting model f. We use a sequence-to-sequence (seq2seq) model (Sutskever et al. (2014)) to map $X^i_{t+1:t+T}$ to $X^i_{t+T+1:t+T+\tau}$ for each series i. Seq2seq is usually a recurrent model. However, when we have a graph structure between the time series, we can exploit it to handle all the time series simultaneously, unlike the usual mechanism that treats each time series separately.

For each time step t', the seq2seq model takes in $X_{t'}$ for all series and uses it to update the internal hidden state from $H_{t'-1}$ to $H_{t'}$. The encoder part of seq2seq performs recurrent updates from $t' = t+1$ to $t' = t+T$, so that H_{t+T} is produced as a summary of the input. The decoder part, on the other hand, uses H_{t+T} to continue the recurrence and evolves the hidden state for another τ steps. Each hidden state $H_{t'}$, $t' = t+T+1 : t+T+\tau$, serves both as the output $\hat{X}_{t'}$ and the input to the next time step.

The recurrence that takes inputs and updates hidden states collectively for all time series uses a graph convolution to replace the usual multiplication with a weight matrix. From

the many possible architectures, e.g., GCRN (Seo et al. (2016)), STGCN (Yu et al. (2018)), and T-GCN (Zhao et al. (2019)), we select the diffusion convolutional GRU (DCGRU) defined in DCRNN (Li et al. (2018b)) because it is designed for directed graphs:

$$R_{t'} = \text{sigmoid}\left(W_R \star_A \left[X_{t'} || H_{t'-1}\right] + b_R\right) \quad C_{t'} = \tanh\left(W_C \star_A \left[X_{t'} || \left(R_{t'} \odot H_{t'-1}\right)\right] + b_C\right)$$
$$U_{t'} = \text{sigmoid}\left(W_U \star_A \left[X_{t'} || H_{t'-1}\right] + b_U\right) \quad H_{t'} = U_{t'} \odot H_{t'-1} + (1 - U_{t'}) \odot C_{t'},$$

where the graph convolution $\star_A$ is defined as

$$W_Q \star_A Y = \sum_{k=0}^{K} \left(w_{k,1}^Q \left(D_O^{-1} A\right)^k + w_{k,2}^Q \left(D_I^{-1} A^T\right)^k\right) Y,$$

with D_O and D_I serving as the out-degree and in-degree matrix and $||$ being concatenation along the feature dimension. Furthermore, $w_{k,1}^Q$, $w_{k,2}^Q$, b_Q for $Q = R, U, C$ are model parameters and the diffusion degree K is a hyperparameter.

Shang et al. (2021) point out that this GNN model can be replaced by similar ones, e.g., T-GCN, without changing the forecast performance too much. In contrast, the more crucial part of the model is the structure learning component presented above, without which it reduces to a model either using no graphs or needing a supplied one, the performance deteriorating in both cases.

Training, optionally with a priori knowledge of the graph

The base training loss per window is the mean absolute error between the forecast and the ground truth

$$\ell_{\text{base}}^t\left(\hat{X}_{t+T+1:t+T+\tau}, X_{t+T+1:t+T+\tau}\right) = \frac{1}{\tau} \sum_{t'=t+T+1}^{t+T+\tau} |\hat{X}_{t'} - X_{t'}|.$$

Moreover, Shang et al. (2021) employ a regularization that enhances graph quality by injecting a priori knowledge of the pairwise interaction into the model. Sometimes a graph among the time series is known, as in the case of the above-mentioned transport network. Even if an explicit structure is unknown, a neighborhood graph such as a kNN graph can still serve as reasonable knowledge. If k is small, the use of kNN encourages sparsity, which bypasses the problem of ℓ_1 constraints that cannot be easily imposed because the graph is not a raw variable to optimize. The cross entropy between θ and the a priori graph A^{a} is

used as the regularization:

$$\ell_{\text{reg}} = \sum_{ij} -A^{\text{a}}_{ij} \log\left(\theta_{ij}\right) - (1 - A^{\text{a}}_{ij}) \log\left(1 - \theta_{ij}\right). \tag{3.45}$$

This results in an overall training loss of $\sum_t \ell^t_{\text{base}} + \lambda \ell_{\text{reg}}$, where $\lambda > 0$ is the regularization magnitude.

4 Research project: Graph Neural Networks with Economic Applications

Armed with solid basics and a well-grounded understanding of GNNs, we can now venture into our research project. It goes by the name of *Graph Neural Networks with Economic Applications* and is carried out by the Chair of Statistics with Application in Business Administration at University of Hamburg together with the pharmaceutical company Novartis. Novartis provides various datasets for this purpose. Along with additionally obtained data, the following four tasks are to be completed.

1. Determination of influencers in Sweden for the breast cancer (BC) and melanoma indications
2. Node classification: comparison of GNN performance with baseline performance
3. Link prediction: comparison of GNN performance with baseline performance
4. Time series forecasting: simultaneous forecasting of multiple sales time series followed by a comparison of GNN performance with baseline performance

From the above tasks, it already becomes apparent that this research project has a strong exploratory focus and is primarily destined to assess the performance of GNNs against baselines on the data given to us.
This section offers a thorough documentation of the research project. In a bid to present the project in an understandable way, we start with a description of the project data and then explain how the data is processed. After that, we elucidate how the HCP graph, which forms the basis for the second and third task, is generated. Subsequently, the four tasks described above are completed.

4.1 Data description

In this subsection, we describe all the data at our disposal for this project and indicate how we obtained it. First of all, the origin of the project data can be divided into the following three categories.

- *Novartis data*: This is internal company data provided to us by Novartis specifically for this project. It cannot be accessed from other sources.
- *Publicly available data*: This includes data directly available on the Internet, e.g., in public databases.[9]
- *Web scraping data*: The data in question was obtained by web scraping.[10]

In the remainder of this subsection, the data originating from these three sources is presented and described. For more insight, including the individual variables of the datasets, look into the respective data processing notebooks containing the data processing steps, which are also the subject of closer consideration in the next subsection.

4.1.1 Novartis data

The Novartis data category contains the following datasets. We explicitly point out that these are data sets that can only be accessed by Novartis, i.e., there is no public access to them.

- `BC & Melanoma targets Sweden.xlsx`: contains the Swedish HCPs that Novartis considers targets for drug sales, along with their doctor or nurse status, their hospital, their function as BC or melanoma HCP, their tier, and their segment. HCPs can occur as both a BC and a melanoma HCP at the same time.
- `Calls data BC & Melanoma 2020-2021.xlsx`: consists of two sheets, of which we use only `report1631874037203`, because it contains the raw data. This sheet contains the calls that Novartis made to Swedish HCPs from January 2020 through September 2021, including the HCP's name and function, the name and address of the associated health care organization, the call focus, and the call method. Only some of the calls were made for advertising purposes.

[9] It should be noted that this may also include data that has been provided to us by Novartis but is also publicly available and therefore does not fall into the Novartis data category.

[10] Although this data is, strictly speaking, also publicly available, it is not downloadable, unlike data in the publicly available data category. Moreover, without methods such as web scraping, it is virtually impossible to obtain it from the web in this richness and in this ordered format.

- `Patients Melanoma + BC market Updated.xlsx`: contains three sheets, of which we only use `Sheet3`, because it gives us the raw data. It contains the number of patients estimated based on packs sold from September 2019 to August 2021 in the various Swedish BC and melanoma regions[11] treated with the 6 medicines Kisqali, Ibrance, Verzenios, Tafinlar-Mekinist, Braftovi-Mektovi, and Zelboraf-Cotellic[12]

- `Sales Melanoma + BC market.xlsx`: includes two sheets, where we only use `Sheet2`, as it provides the raw data. This sheet contains the sales in Swedish kronor from September 2019 to August 2021 in the various Swedish BC and melanoma regions for the 6 medicines Kisqali, Ibrance, Verzenios, Tafinlar-Mekinist, Braftovi-Mektovi, and Zelboraf-Cotellic.

- `20211229_IHME_GBD2019.xlsx`: offers actual data on the number and rate of deaths, prevalences, and incidences in Sweden for various types of breast cancer and skin cancer for men and women from 2012 to 2019.

- `20211229_IHME_Forecast_Data.xlsx`: contains various forecasts of the number and rate of deaths, prevalences, and incidences in Sweden for different types of breast cancer and skin cancer for men and women from 2012 to 2030.

- `Mapping Bricks to Region.xlsx`: assigns to each brick in Sweden the associated BC and melanoma region defined by Novartis and its population size.

- `Sweden Population by brick 2022.xlsx`: assigns to each brick in Sweden the associated county and its population size in 2022.

4.1.2 Publicly available data

Four datasets below fall into this category. Of these, the first three were provided by Novartis, but can be downloaded from the websites of two Swedish institutes. The fourth dataset was incorporated into the project by us and comes from a GitHub repository.

[11] For additional information on Swedish BC and melanoma regions and a map of Sweden clarifying their respective location, see appendix B.

[12] Of the 6 medicines, 3 are targeted at BC patients and 3 are targeted at melanoma patients.
Kisqali is a drug by Novartis for the treatment of women with hormone receptor-positive breast cancer that has metastasized or progressed locally. Ibrance, on the other hand, is by Pfizer and is intended for the treatment of women with advanced hormone receptor-positive breast cancer. Verzenios by Lilly is for the treatment of women with hormone receptor-positive breast cancer that has metastasized or is locally advanced.
Tafinlar-Mekinist by Novartis, Braftovi-Mektovi by Pfizer and Zelboraf-Cotellic by Genentech are all used to treat adults with black skin cancer, i.e., a malignant melanoma, that can no longer be operated on or has already metastasized.

- `Population by Age 2020.xlsx`: contains the population by Swedish counties, age and sex on November 1, 2020. The data was downloaded from the Statistics Sweden website (Statistics Sweden (2021)).
- `Statistikdatabasen_Breast Cancer_17_09_2021 14_15_07.xlsx`: gives the number of new breast cancer cases - irrespective of tumor type - in the age group 0-85+ from 1970 to 2019 by sex and Swedish county. The data was downloaded from the Socialstyrelsen website (Socialstyrelsen (2021a)).
- `Statistikdatabasen_Malignant Melanoma 17_09_2021 14_13_18.xlsx`: includes the number of new malignant melanoma of skin cases - irrespective of tumor type - in the age group 0-85+ from 1970 to 2019 by sex and Swedish county. The data was downloaded from the Socialstyrelsen website (Socialstyrelsen (2021b)).
- `svenska-stader.csv`: contains for each locality in Sweden the corresponding municipality and county as well as the latitude and longitude. The data was downloaded from the GitHub repository svenska-stader by Kron (2020).

4.1.3 Web scraping data

For the node classification and the link prediction tasks, we need an HCP graph, i.e., a graph whose nodes are given by the different HCPs and whose links represent the connections between the HCPs. From the data described above, a link between two HCPs can only be inferred via the hospitals at which the two HCPs work. If two HCPs work at the same hospital, it can be assumed that they are linked. Nevertheless, a connection between two HCPs can also be established by other circumstances, such as the existence of joint publication. However, such information is not included in the above data.
For this reason, we would like to know for each HCP which papers they have published and with whom. The Internet offers various sources from which such data can be obtained. A manual collection of all papers of an HCP together with all co-authors is impracticable and error-prone. Yet, *web scraping* offers the possibility to collect this data automatically. Web scraping is a method in which the content of interest can be read out and extracted from web pages with the help of a code adapted to one's own requirements.
For our goal of finding papers and co-authors for all HCPs, PubMed and Google Scholar are two suitable sources from which to scrape this data. PubMed is a database specialized in medical articles, while Google Scholar is for general literature searches for scientific articles. Both of them purvey a multitude of useful papers to us.

PubMed

In order to scrape papers and co-authors from PubMed for each HCP[13], we use the Python library PyMed (Wobben (2020)), which provides access to PubMed data through the PubMed API. The library performs the following tasks: It queries the PubMed database, batches the requests for better performance as well as parses and cleans the retrieved papers. After importing and instantiating the PubMed class, we can scrape directly without any later issues. For each HCP we only need to pass to the query their name as well as the maximum number of papers to scrape for them. We include all the code for scraping PubMed in the notebook `web_scraping_pm.ipynb`. Our code is written in such a way that we use a loop to scrape a maximum of 100 papers for each HCP. Eventually, we get the following dataset.

- `results_queries_pm.csv`: contains row by row for each paper scraped from PubMed the authors, journal, keywords, publication date, title and other variables. For HCPs for whom no papers were scraped, there is always one row featuring "NaN" as the value for all variables. In total, this dataset has 4147 rows and 13 columns.

Google Scholar

The principle of scraping from Google Scholar is similar. To our disadvantage, however, scraping from Google Scholar entails some difficulties and is therefore many times more complex than scraping from PubMed. In order to scrape papers and co-authors from Google Scholar for each HCP, the Python library Beautiful Soup (Richardson (2020)) can be leveraged together with the Python library Requests (Reitz (2022)). Beautiful Soup is generally useful for pulling data out of HTML and XLM files, and thus is not specific to scraping Google Scholar.[14] The way scraping works is as follows. Requests sends a GET request to the Google Scholar url with the following components as query string: a dictionary of HTTP headers, a dictionary of the protocol to the proxy url, a parameter dictionary with the name of the respective HCP, the language of Google Scholar and the index of the

[13] It should be noted that we do not consider the BC or melanoma indications when scraping. Thus, we perform scraping only for all physically distinct HCPs. This means that we scrape for an HCP only once even if they are active for both BC and melanoma indications.

[14] A very helpful introduction specifically to scraping Google Scholar using Beautiful Soup is offered by Zub (2021). For further study, Vanden Broucke and Baesens (2018) provide some code examples for scraping with Beautiful Soup in chapters 3.4 and 3.5 of their book (pp. 61-77). In addition, Heydt (2018) gives an example-based training on how to take advantage of Beautiful Soup in scraping.

first of the 10 articles to be scraped. This calls Google Scholar for the respective HCP. Beautiful Soup then reads out the requested information. This is done by having Beautiful Soup run a CSS selector over the ".gs_ri" tag after parsing the web page with our chosen lxml parser, thereby getting all the papers displayed for an HCP. After that, CSS selectors with associated tags are run over each scraped paper again to extract the title, the title link, the authors, the publication info, i.e., the journal and the date where possible, and the snippet, i.e., the extract, for each paper.

However, as mentioned above, we also encounter some difficulties that we have to work around again and again when scraping Google Scholar with Beautiful Soup. Here, the following three problems are worth highlighting.

First, for many HCP, papers are scraped that do not belong to them at all. This problem is particularly salient when specifying the respective HCP's name in the parameter dictionary without quotation marks. We can relatively easily solve this problem by putting the name of the HCP in quotes in the parameter dictionary.

Secondly, only the first ten papers of an HCP are scraped without any further modification of the code. This is because only papers appearing on the first page for an HCP in Google Scholar are scraped. If we want to scrape more than a maximum of ten papers for an HCP, we have to add an additional key "start" to the parameter dictionary and set the associated value of that key to e.g. 10 to get papers 11 to 20 for an HCP, with indices from 10 to 19.[15] We aim to scrape a maximum of 30 papers for an HCP. Therefore, we have to perform scraping three times for each HCP by altering the value for "start" every time. First, we set "start" to 0 for papers 1 to 10, then to 10 for papers 11 to 20 and finally to 20 for papers 21 to 30. So, we perform scraping in blocks of 10 papers at a time.

Thirdly, the face the problem that Google Scholar notices that queries repeatedly come from the same IP address in very short intervals. For this reason, Google Scholar blocks the user's IP address if - as in the case of scraping PubMed - the user is scraping for all HCPs one right after the other. In order to achieve that we are able to scrape papers for all HCPs, we take two expedient measures. On the one hand, instead of scraping all papers for all 208 HCPs in direct succession, we scrape them in 16 blocks of 13 HCPs, with at least 90 minutes elapsing after each block before scraping for the next block. After some experimentation, it turns out that these 16 blocks of 13 HCPs each enable problem-free scraping of the papers for the 13 HCPs. On the other hand, after scraping all papers for one HCP, a random pause of the code of 1 to 5 seconds is enforced to make

[15] Without further adjustment the default value for "start" is set to 0, so that the first ten papers, with indices from 0 to 9, are scraped.

Google Scholar think that the pauses between the queries are random instead of systematic. Scraping in 16 blocks with 13 HCPs each - as described above - yields the following results: The notebook `web_scraping_gs_part1-4.ipynb` scrapes the papers for blocks 1-4, the notebook `web_scraping_gs_part5-8.ipynb` scrapes the paper for blocks 5-8, and the notebook `web_scraping_gs_part9-12.ipynb` scrapes the paper for blocks 9-12. The notebook `web_scraping_gs_part13-16.ipynb` scrapes the papers for blocks 13-16. The datasets `results_queries_gs_1.csv` through `results_queries_gs_16.csv` contain the scraped papers for blocks 1 through 16. These 16 datasets are stacked together at the end of the notebook `web_scraping_gs_part13-16.ipynb`. In the end, this results in the following dataset.

- `results_queries_gs.csv`: contains row by row for each paper scraped from Google Scholar the authors, the publication information, i.e., journal and publication year where possible, the title, and other variables. For HCPs for whom, for instance, no papers could be scraped, this dataset always contains three rows, where all variables take "NaN" as value. The reason is that no papers could be scraped in any of the 3 scraping steps performed for an HCP. In total, this dataset has 3778 rows and 7 columns.

Of course, the code included in the above notebooks runs through every time without any problems. However, since the results of scraping always depend on the exact point in time of scraping, the datasets `results_queries_pm.csv` and `results_queries_gs.csv` can no longer be reproduced exactly with the given code, since they only contain the papers that could be scraped from PubMed and Google Scholar on 31/01/22. Due to new papers published in the meantime, repeated scraping may result in more papers.

4.2 Data processing

After presenting the project data in its original form, we now outline how we process the above datasets to make them usable for our four later tasks. It should be noted that we process all datasets described in the previous subsection, even if we do not make use of them in later tasks. Our intention to process these datasets nonetheless is attributable to the fact that we would like them to be in such a condition that they are readily usable for further tasks in the future.

Data processing is divided into two steps: data preparation and creation of new datasets. The data preparation includes steps like renaming variables, deleting irrelevant variables,

converting variables to the correct data type, reshaping datasets, e.g., from wide to long format, but also steps like generating new variables for a dataset. The creation of new datasets, by contrast, is about generating new datasets from several other datasets.

4.2.1 Data preparation

We now explain step by step how we prepare the individual datasets from the preceding subsection. Below you can see which notebooks we use for our data preparation. For each notebook we state which dataset is prepared in it, which preparation steps are performed and what the prepared dataset is called. Consult the corresponding notebooks for the relevant code and further details.

- `0_targets_preparation.ipynb`: In this notebook, the preparation of the dataset `BC & Melanoma targets Sweden.xlsx` is carried out. The last rows are removed because they are irrelevant, the columns are renamed, irrelevant variables are dropped and the variables are converted to the correct datatype. A few values of the variable "top_account_name" are corrected. The variables "address", "locality", "municipality" and "territory" are also added using the data sets `svenska_stader.pkl` and `mapping.pkl`, which are explained in more detail below. For one HCP, the value of the variable "tier" is corrected. The prepared dataset is given by `targets.pkl`.

- `1_calls_preparation.ipynb`: This notebook contains the preparation of the sheet `report1631874037203` in `Calls data BC & Melanoma 2020-2021.xlsx`. Due to irrelevance we remove the last rows, we also rename the columns, drop irrelevant variables and convert the variables to the correct data type. Finally, we save the prepared dataset as `calls.pkl`.

- `2_patients_preparation.ipynb`: We prepare Sheet3 in `Patients Melanoma + BC market Updated.xlsx` in this notebook by deleting the first row and first column due to irrelevance, renaming the columns, dropping irrelevant variables and converting the variables to the correct data type. Then we prepare the patients values: negative values are replaced by 0, NaNs are replaced by 0 and all values are rounded to the nearest integer - as discussed with Novartis. We also remove rows for the regions "SE-other" and "SE-other ONCO".[16] Furthermore, we reshape the dataset from wide to

[16] "SE-other" and "SE-other ONCO" are no physical regions, but represent the online pharmacy related to the BC and melanoma indications. After consultation with Novartis, we exclude them from the set of BC and melanoma regions.

long format, convert the new variables to the correct data type and subsequently sort the dataset. Finally, we save the prepared dataset as `patients.pkl`.

- `3_population_preparation.ipynb`: In this notebook, the dataset `Population by Age 2020.xlsx` is prepared. For this we remove some rows at the beginning and end of the dataset, rename the columns, convert the variables to the correct data types and we fill forward the variables "region" and "age". The prepared dataset is `population.pkl`. It is not used in our later tasks, but might be useful in the future for other questions to be investigated.

- `4_sales_preparation.ipynb`: The preparation of `Sheet2` in `Sales Melanoma + BC market.xlsx` takes place in this notebook. To do this, we remove the first two rows and the first column of the dataset, rename all columns, and drop irrelevant variables. Then we remove the brackets around some sales values and replace them with a minus. Next, we convert the variables to the correct data type. As with the patients values, we then replace negative values and NaNs with 0 and round all values to the nearest integer. In addition, we remove the regions "SE-other" and "SE-other ONCO". Unlike in the patients dataset, however, the sales values for the melanoma indication are not yet available in the respective combinations Tafinlar-Mekinist, Braftovi-Mektovi and Zelboraf-Cotellic, but only individually. Therefore, we create the above-mentioned combinations from the corresponding sales of the individual drugs. Hereafter, we reshape the dataset from wide to long format, rearrange the columns, convert the new variables to the correct data type and finally sort the dataset. Lastly, we save the prepared dataset as `sales.pkl`.

- `5_stats_bc_preparation.ipynb`: Here we provide the preparation of the dataset `Statistikdatabasen_Breast Cancer_17_09_2021 14_15_07.xlsx`. The header is replaced by the first row, two rows are removed, the columns are renamed and the variables are converted to the correct data type. In what follows we reshape the dataset from wide to long format, convert the new variables to the correct datatype and finally sort the dataset. The prepared dataset is saved as `stats_bc.pkl`.

- `6_stats_me_preparation.ipynb`: This notebook gives the preparation of the dataset `Statistikdatabasen_Malignant Melanoma 17_09_2021 14_13_18.xlsx`. First, we replace the header with the first row and remove two rows before renaming the columns and converting the variables to the correct data type. Then we reshape the dataset

from wide to long format and convert the new variables to the correct data type. At last, we sort the dataset and save it as `stats_me.pkl`.

- `7_mapping_preparation.ipynb`: We perform the preparation of the dataset `Mapping Bricks to Region.xlsx` in this notebook. For this we rename the columns, drop irrelevant variables, convert the variables to the correct data type and remove rows where the variable "brick" has the value "999 Unknown" or "99 Unknown". To finish, we save the prepared dataset as `mapping.pkl`.

- `8_population_by_brick_preparation.ipynb`: This notebook contains the preparation of the dataset `Sweden Population by brick 2022.xlsx`. We rename the columns, drop irrelevant columns, convert the variables to the correct data type and remove the rows where the variable "brick" has the value NaN or "99 Unknown". Ultimately, we save the prepared dataset as `population_by_brick.pkl`.

- `9_ihme_forecasts_preparation.ipynb`: Here we provide the preparation of the dataset `20211229_IHME_Forecast_Data.xlsx`. We do this by renaming the columns, dropping irrelevant variables, converting the variables to the correct data type, and sorting the dataset. We save the prepared dataset as `ihme_forecasts.pkl`.

- `10_ihme_actuals_preparation.ipynb`: In this notebook, we prepare the dataset `20211229_IHME_GBD2019.xlsx` by renaming the columns, dropping irrelevant variables, converting the variables to the correct data type, and sorting the dataset. Then we save the prepared dataset as `ihme_actuals.pkl`.

- `11_svenska_stader_preparation.ipynb`: This notebook offers the preparation of the dataset `svenska-stader.csv`. The preparation consists of renaming the columns and dropping irrelevant variables. We save the prepared dataset as `svenska_stader.pkl`.

- `12_results_pm_preparation.ipynb`: The dataset `results_queries_pm.csv` is prepared in this notebook. However, the preparation of the PubMed data is many times more complex and challenging than the preparation of the previous datasets. Our procedure is described below.

  First, we get an overview of where in the dataset there are missing values for the variables of interest: "authors", "journal", "keywords", "publication_date" and "title". We find that there are 45 missing values each for "authors", "journal", "keywords" and "publication_date", while there are 68 missing values for "title". The 45 missing

values are rows for HCPs for whom no paper could be scraped.[17]. For the remaining 23 missing values in "title" it is not clear where they come from. A closer manual inspection in PubMed reveals that there are 23 papers for which only their title could not be scraped due to containing an italic string, a superscript, a bold string or special characters such as "å", "ñ" or "Å".

We then proceed to delete the 45 rows for the HCPs without papers. We keep the other 23 rows containing the papers whose titles we do not have in the dataset. In the next step, we drop irrelevant variables, leaving only "hcp_name", "num_articles", "abstract", "authors", "journal", "keywords", "publication_date" and "title" as variables. Besides, we check if there are duplicates. This is not the case.

We now turn to the variables of the dataset individually and prepare them: For "hcp_names", we observe that the names of the HCPs contained in it do not have a regular format. Some names consist of only a first name and last name, while other names have two first names and a last name, or one first name and two last names. We choose to keep only the first letter of the first name and the entire last name so that the names have a uniform format.[18] It should be noted here that hyphenated names, no matter if first names or last names, are considered as one name.

The most time-consuming and demanding part of the preparation of this dataset is the preparation of "authors". The authors of a paper are given as a list of tuples whose values are enclosed in single quotes. Although regularly the first value of the tuple is the first name and the second value of the tuple is the last name, this is not always the case. The entire name may be given as the second value of the tuple and the first value of the tuple may be "None". Further irregularities can also occur when names contain an apostrophe, so that the corresponding name is no longer enclosed in single but in double quotes. In order to obtain the authors from the lists of tuples with the described irregularities, extensive string processing is necessary, which requires the use of 13 consecutive replacement commands among other processing steps. After the authors are available, their individual names are then truncated according to the convention described above. Only the initial letter of the first name and the entire last name are preserved.

[17] As mentioned in the previous subsection, for HCPs for whom no paper could be scraped, there is one row in the dataset containing "NaN" as value for each variable except "hcp_name" and "num_articles".

[18] The reason for this decision is the following. With a view to our later graph generation, we need to achieve uniformity of names in PubMed and Google Scholar. Names scraped from Google Scholar already have exactly this format.

We find that when scraping from PubMed, papers can even be scraped for HCPs who have no papers on PubMed. After preparing "hcp_name" and "authors", we can now delete those papers from the dataset for which the respective HCP does not appear among the authors.
Preparing the variable "abstract", we want to remove the many stopwords, i.e., the words that are necessary to form a sentence, but which contribute little to the meaning of a sentence, such as "is", "has", "on" or "to". We also remove special characters and convert the abstracts to lower case.
The preparation the variable "title" is similar: we remove punctuation and stopwords and convert the titles to lower case.
For the variable "journal", we only convert to lower case.
With the variable "keywords" we notice that the keywords are given as a string of a list, whose entries are set in single quotes. However, the entries may also be enclosed in double quotes if a keyword contains an apostrophe. We prepare the variable keywords in such a way that we remove the square brackets, the single quotes and the double quotes, so that only keywords separated by commas remain in the end. We save all the different keywords scraped from PubMed in the list `keywords_list.txt`. The reason for this is that we later want to use these keywords to filter out those papers that have no medical background from the papers scraped from Google Scholar. For this purpose, we use the keywords of the papers scraped from PubMed, because PubMed only provides medical papers. Note, however, that the keywords may include a smaller number of non-medical words, such as "education", "vocabulary" or "young".
For the variable "publication_date", we keep only the year of publication and rename the variable to "publication_year".
In addition, we prepare the variable "num_articles" by updating it. In the course of our previous preparatory steps, some papers were deleted from the dataset and thus the number of papers of an HCP may have changed. So, to capture the current number of papers for the HCPs, we update "num_articles".
Finally, we add a new variable "scraped_from", which indicates for all papers in the dataset that they have been scraped from PubMed. Thus, it always takes the value "pm".

We save the prepared dataset as `results_queries_pm_cleaned.csv`. Note that not all variables of this dataset will be used in later tasks. Notwithstanding, we prepare

them in order to be able to use the different variables in further analyses in the future, if required.

- `13_results_gs_preparation`: The dataset `results_queries_gs.csv` is prepared in this notebook. Just as the preparation of the PubMed data, the preparation of the Google Scholar data is considerably more laborious than the preparation of the previous datasets. We adopt the following procedure.

  First, we drop irrelevant variables, i.e., in this case only "title_link". We continue by examining where there are missing values. We find out that the variables "title", "authors" and "publications" have 214 missing values, while the variable "snippet" has 258 missing values. The 214 missing values are due to HCPs where either no paper or not more than 10 or 20 papers could be scraped. We delete the corresponding rows. However, we keep the 44 rows with missing values for "snippet". We then check whether there are duplicate rows. In total, there are 174 duplicate rows, which we also delete from the dataset.

  Our focus is now on the individual variables of the dataset and their preparation. With the variable "title", we notice that there are often different string components which feature in front of the title, but do not belong to the title, e.g., "[BOOK][B]". We also want to remove the string "\xa0...", which occurs frequently. Likewise, we aim to remove punctuation and stopwords and convert the titles to lower case.
  Having performed these steps, we use the above-mentioned keywords from PubMed to filter out papers from our dataset that do not have at least one of these keywords in their title.
  Next, we turn to the variable "hcp_name". To preserve the convention for names described above, we again only keep the first letter of the first name and the entire last name. Hyphenated first names or last names are again considered as one first name or last name, analogous to above. We store the truncated names in the new variable "hcp_name_short". Furthermore, we keep only the first and the last name of an HCP in the variable "hcp_name", where hyphenated names are again considered as one name.
  The variable "authors" contains the authors as a string, usually in single quotes, with the individual names separated by a comma followed by a space. The strings end either with "...\xa0", "...", "xa0" or a space. If the string contains a name with apos-

trophe, the whole string is enclosed in double quotes. So, we now prepare the variable "authors" by removing "...\xa0", "...", "xa0" and the spaces at the end of a string. We also remove the apostrophes. Afterwards, we truncate for each author his name according to the above-mentioned convention.
To make sure that we do not have any papers left in the dataset that do not belong to the HCP to which they should belong, we delete the papers from the dataset where the respective HCP is not among the authors, as we did with the PubMed data.
The variable "publications" sometimes contains the journal and year of publication for a paper, sometimes only one of them and sometimes none of them at all. Our goal is to split this variable into the two new variables "journal" and "publication_year", which contain the respective journal and year of publication for the respective paper, if these are given. The original variable "publications" is deleted.
To prepare the variable "journal", we convert to lower case.
We rename the variable "snippet" to "abstract" for consistency with the PubMed data. Furthermore, the stopwords, the often occurring "..." string and special characters are removed for the variable "abstract". We also convert the abstracts to lower case.
In order to still have the correct number of papers for each HCP after deleting papers from the dataset, we update the variable "num_articles". Finally, we add a new column "scraped_from" indicating for all papers in the dataset that they were scraped from Google Scholar. Therefore, it always takes the value "gs".

We save the prepared dataset as `results_queries_gs_cleaned.csv`. As with the PubMed data, not all variables of this dataset are used in our tasks. Their preparation serves the purpose of facilitating their use in further tasks in the future, if required.

4.2.2 Creation of new datasets

After dealing with the preparation of individual datasets, we now take a look at the creation of new datasets using several of the previously prepared datasets. In the following, we disclose which notebooks we use for the creation of new datasets, which of the previous datasets we leverage and how we create new datasets from them.

- `0_aggregation_population_by_region.ipynb`: In this notebook, we merge the population for each of the bricks in the dataset `population_by_brick.pkl` to the bricks and Swedish BC and melanoma regions in the dataset `population_by_brick.pkl` to get the population for each Swedish BC and melanoma region. We aggregate the

population numbers over bricks belonging to the same BC and melanoma region and finally obtain the dataset `population_by_region.pkl`. It contains the corresponding population for each Swedish BC and melanoma region.

- `1_number_hcps_hospitals_calls.ipynb`: Here we want to determine the number of HCPs and hospitals in each BC and melanoma region. Also, for each month between 2020-01 and 2021-08, we would like to determine the number of calls made to HCPs in each BC and melanoma region.

  First, we take the dataset `targets.pkl` and correct inconsistencies in the variable "top_account_name". Then we drop irrelevant variables, leaving us with the variables "account_name", "top_account_name" and "indication", to which we can merge the hospital's city in the new variable "locality" using a dictionary that assigns the corresponding address to each hospital in "top_account_name". Using `svenska_stader.pkl`, we can now also assign the county associated with the city in the new variable "municipality". However, care must be taken here, as some assignments are incorrect[19], so that we must correct them manually. Then, using the dataset `mapping.pkl`, we can aggregate the data in order to create two new datasets, `hcp_counts.pkl` and `hospital_counts.pkl`, which give us the corresponding number of HCPs and hospitals in each BC and melanoma region.

  Next, we take the dataset `calls.pkl` and filter it so that only promotional calls made from 2020-01 to 2021-08 remain. Again, using both a dictionary that assigns addresses to hospitals and the dataset `svenska_stader.pkl`, we add the variables "locality" and "municipality" and correct any incorrect assignments manually. Using the dataset `mapping.pkl`, we now create the promotional calls made to HCPs on a monthly basis between 2020-01 and 2021-08 in each BC and melanoma region and save them in the new dataset `call_counts.pkl`.

Thanks to the above preparatory steps, we are now equipped with all the data we need for our next tasks. We can use it right in the upcoming subsection to generate our HCP graph.

4.3 Graph generation

Having completed data processing, we now elaborate on the steps we go through to generate the HCP graph below. Beyond that, we also explain how we get the node features and the edge features for the next tasks.

[19] As mentioned above, `svenska_stader.pkl` is not consistently error-free.

To be able to generate our HCP graph in its basic form - consisting of nodes and edges only - we need a list of all the edges[20] that exist between two nodes in the graph. Since we intend to use the HCP graph later for node classification and link prediction, we also need the corresponding node features for each node in the graph. In order to be able to trace at any time what kind of connection an edge between two nodes represents, it is also useful to have the edge features for each edge. It should be borne in mind that we do not use the edge features in our future learning tasks of this project. Thus, we work with a homogeneous graph. However, given the edge features, a heterogeneous graph could also be created.
In the following, we show how we create the edge list, the node features data frame and the edge features data frame. By far the most time-consuming part is the extraction of the edges, whereas the creation of the node features and edge features requires comparatively little effort. Finally, the HCP graph can be created, which we plot at the end of the subsection and briefly analyze using a summary of its most important properties. The relevant code for graph generation is provided in `graph_generation.ipynb`.

4.3.1 Edge extraction

There are different edge types on the basis of which two HCPs can be connected. Specifically, we connect two HCPs with an edge if at least one of the following three conditions is met.

- The two HCPs have published at least one paper together and are thus co-authors of at least one paper. In such a case, it can be assumed that the HCPs have already worked together in a close context. Accordingly, the HCPs know each other well and such an edge can be considered strong.
- The two HCPs have published at least one paper in the same journal so far. The idea is that HCPs know the journals they publish in very well and are also well informed about other HCPs publishing in the same journal. Thus, the two HCPs belong to the same scientific community but do not need to know each other personally. An edge extracted in this way should be considered weak.
- The two HCPs work in the same hospital and thus work with each other on a daily basis. For this reason, the HCPs know each other very well, which means that such an edge should be interpreted as strong.

To extract these edges, we use the prepared data frames `results_queries_pm_cleaned.csv` and `results_queries_gs_cleaned.csv` for the first two edge types as well as `targets.pkl`

[20] We want to create a graph consisting of only one connected component. In such a case, a list of nodes is not needed because it is implicitly given by the list of edges.

for the third edge type since it contains the respective hospital for an HCP. Edge extraction proceeds in the following steps.

1. `results_queries_pm_cleaned.csv`: We use the PubMed data for both edges by joint paper and edges by joint journal.[21]

   First, we extract edges by joint paper. For each HCP for whom we scraped at least one paper from PubMed[22], we create a list containing all co-authors. Among the co-authors, there may be other HCPs from our targets dataset, but most of them are not among the HCPs from the targets dataset. From this list, we create an edge list - for each HCP for whom we scraped at least one paper from PubMed - that includes the edges to all co-authors exactly once. We use this edge list to build the `DN_pm_papers` graph using networkx (Hagberg et al. (2008)). Of course, we only want to keep edges that exist between two HCPs from the targets data frame and exclude other co-authors from consideration. Using the `DN_pm_papers` graph, we create a dataset that contains, row by row, the HCPs for whom we scraped at least one paper from PubMed, and for these, the one-hop neighbors[23], two-hop neighbors[24] and one-two-hop neighbors[25]. Based on this, we can then create the two graphs `DN_pm_paper_1h` and `DN_pm_paper_1h2`. In the following, we continue working only with the nodes and edges from `DN_pm_journal_1h`, i.e., we only consider direct edges hereafter.[26].

   After that, we extract edges by joint journal. We make a list for each HCP for whom we scraped at least one paper from PubMed with all the journals that this HCP has published in, and then we make a list - for each HCP for whom we scraped at least one paper from PubMed - with all other HCPs - for whom we scraped at least one paper from PubMed - who have published in the same journals. We then make an edge list for each HCP - for whom we scraped at least one paper from PubMed - which contains all edges exactly once. We use this edge list to build the graph `DN_pm_journals` using networkx (Hagberg et al. (2008)). Using the `DN_pm_journals` graph, we create a dataset that contains, row by row, the HCPs - for whom we scraped at least one paper from PubMed - and for these, the one-hop neighbors, two-hop neighbors and one-two-hop neighbors. We can then build the graphs `DN_pm_journal_1h` and

[21] In the following, we call an edge of the first edge type "edge by joint paper" and an edge of the second edge type "edge by joint journal".
[22] Strictly speaking, these are HCPs for whom at least one paper remains after cleaning the PubMed data.
[23] One-hop neighbors are HCPs to which there is a direct edge.
[24] Two-hop neighbors are HCPs who have an indirect edge through another HCP.
[25] One-two-hop neighbors are HCPs that are at least either one-hop neighbors or two-hop neighbors.
[26] For further analysis and questions in the future, edges to two-hop neighbors for each HCP can of course also be included in the consideration and thus in the final graph.

`DN_pm_journal_1h2h`, respectively. These two graphs now contain the one-hop and one-two-hop neighborhood for each of these HCP. In the following, we continue working only with the nodes and edges from `DN_pm_journal_1h`.

2. `results_queries_gs_cleaned.csv`: We use the Google Scholar data for both edges by joint papers and edges by joint journal.

   Before edge extraction, the following three steps must be completed. First, we have to find the journal for four papers, since the respective journals are not or not completely given in the dataset. Second, we filter out papers that we have already scraped from PubMed and for which edges have already been extracted. Third, for each HCP we replace their name - which is given in truncated format, i.e., only the first letter of the first name and the entire last name - in the authors variable with their full name as given in the variable "hcp_name" to be able to extract edges in the same format as in PubMed. However, we must be careful with Christine Lundgren and Per Karlsson. Christine Lundgren's truncated name is C Lundgren and two other HCPs have this truncated name: Carin Lundgren and Claudia Lundgren. All papers from Google scholar where C Lundgren appears among the authors are by Christine Lundgren. Per Karlsson's truncated name is P Karlsson and this truncated name is also given to another HCP: Pernilla Karlsson. All papers where P Karlsson appears among the authors are by Per Karlsson. For these two HCPs, C Lundgren must be manually replaced by Christine Lundgren and P Karlsson by Per Karlsson, after which the remaining truncated names are automatically replaced by the full names. Now our procedure is analogous to that for the PubMed data.

   First, we extract the edges by joint paper. For each HCP for whom we scraped at least one paper from Google Scholar[27], we create a list containing all co-authors. Among the coauthors, there may be other HCPs from our targets data frame, but most of them are not among the HCPs from the targets data frame. From this list, for each HCP - for whom we were scraped at least one paper from Google Scholar - we create an edge list that includes the edges to all co-authors exactly once. We use this edge list to build the `DN_gs_papers` graph using networkx (Hagberg et al. (2008)). Of course, we only want to keep edges that exist between two HCPs from the targets data frame and exclude other co-authors from consideration. Using the `DN_gs_papers` graph, we create a data frame that contains, row by row, the HCPs - for whom we

[27] Strictly speaking, these are HCPs for which at least one paper remains after cleaning the Google Scholar data and performing the steps described in the last paragraph.

scraped at least one paper from Google Scholar - and for these, the one-hop neighbors, two-hop neighbors and one-two-hop neighbors. Based on this, we can then create the two graphs `DN_gs_paper_1h` and `DN_gs_paper_1h2`. In the following, we continue working only with the nodes and edges from `DN_gs_journal_1h`, i.e., we only consider direct links in the following.

After that, we extract edges by joint journal. We make a list for each HCP for whom we scraped at least one paper from Google Scholar with all the journals that this HCP has published in, and then we make a list - for each HCP for whom we scraped at least one paper from Google Scholar - with all other HCPs - for whom we scraped at least one paper from Google Scholar - who have published in the same journals. We then make an edge list for each HCP - for whom we scraped at least one paper from Google Scholar - which contains all edges exactly once. We use this edge list to build the graph `DN_gs_journals`. Using the `DN_gs_journals` graph, we create a data frame that contains, row by row, the HCPs - for whom we scraped at least one paper from Google Scholar - and for these, the one-hop neighbors, two-hop neighbors and one-two-hop neighbors. We can then build the graphs `DN_gs_journal_1h` and `DN_gs_journal_1h2h`, respectively. These two graphs now contain the one-hop and one-two-hop neighborhood for each HCP scraped. In the following, we continue working only with the nodes and edges from `DN_gs_journal_1h`.

3. `targets.pkl`: We use the targets dataset only for edges by joint hospital.

   As a preliminary step, we retain only the first and the last name of the HCP names. All names in between are removed. Thus, the same name format is achieved as for the PubMed and Google Scholar data.

   An edge is only extracted for two HCPs if they work in the same hospital. Thus, in the context of edges by joint hospital, there are only one-hop neighbors.

   For each HCP from the targets dataset, we create a list of all other HCPs working in the same hospital. From this, we create an edge list for each HCP that contains the edges to all other HCPs working in the same hospital. We use all these edges to create the `DN_hospitals` graph using networkx (Hagberg et al. (2008)).

The edges just extracted are now processed further. We combine the edges by joint paper extracted from PubMed and Google Scholar to the list `paper_links`. Based on this list, we create a dataset containing the edges row by row. For the two respective HCPs specified by name, we also assign their indexes from the targets dataset. For HCPs active for both BC and melanoma, each edge becomes two edges: one edge to the BC HCP and another to the

melanoma HCP. Moreover, we also include an edge between the same HCP acting once as a BC and once as a melanoma HCP.

Similarly, we combine the edges by joint journal extracted from PubMed and Google Scholar to the list `journal_links`. Using this list, we create a dataset that again contains the edges row by row. For the two respective HCPs specified by name, we again assign their indexes from the targets dataset. As before, for HCPs active for both BC and melanoma, each edge becomes two edges: one edge to the BC HCP and another to the melanoma HCP. In addition, we again include an edge between the same HCP acting once as BC and once as melanoma HCP.

For the edges by joint hospital given in the list `hospital_links`, we also create a dataset that contains the edges row by row. Just as before, we assign the two respective HCPs their indexes from the targets dataset. Again, for HCPs active for both BC and melanoma, each edge becomes two edges: one link to the BC HCP and another to the melanoma HCP. In addition, we include an edge between the same HCP acting once as a BC and once as a melanoma HCP.

Finally, all the edges extracted so far are combined into the list `all_link_list`, where each edge occurs only once. We end up with a total of 5821 edges. From this, the final graph `DN_all_links` is created.

Table 2 summarizes how many nodes and edges we were able to extract in every single step of the edge extraction process described above. Here, *Only HCPs: 1 hop* - highlighted in bold - is the only part relevant for us.

		PubMed		Google Scholar		Targets
		Paper	Journal	Paper	Journal	Hospital
All co-authors	Nodes	11204	126	1369	95	-
	Edges	17515	3639	1449	741	-
Only HCPs: 1 hop	Nodes	**94**	**126**	**69**	**95**	**204**
	Edges	**441**	**3639**	**78**	**741**	**1069**
Only HCPs: 2 hops	Nodes	123	126	103	95	-
	Edges	2999	7567	357	3037	-

Table 2: Number of extracted nodes and edges

4.3.2 Node features

The targets dataset forms the basis for the creation of the node features dataset. Initially, all variables that become relevant in some form in the future are merged to the targets dataset. For each HCP, this includes the hospital counts from the dataset `hospital_counts.pkl`, the HCP counts from the dataset `hcp_counts.pkl` and the population in each BC and melanoma region from the dataset `population_by_region.pkl`. We then use the dataset `calls.pkl` to determine the number of minutes that each HCP was contacted by Novartis during the period from 01-2021 to 08-2021 through the channels "e-mail", "face to face", "phone", "virtual MS teams", "virtual other", "virtual Veeva engage", "virtual Zoom", and "voice only call". We also merge the resulting 8 variables to the targets dataset. In addition, we determine for each BC and melanoma region how many patients there were and how high the sales were in the period from 09-2019 to 08-2021, and merge these variables to the targets dataset as well. Furthermore, we obtain for each HCP the total number of papers that we were able to scrape for them via PubMed and Google Scholar and merge this variable to the targets dataset as well. We also merge a variable to the targets dataset that specifies the node index of each HCP from 0 to 228. Finally, we merge the five variables "degree", "degree centrality", "eigenvector centrality", "closeness centrality", and "betweenness centrality" to the targets dataset.[28] Note that this dataset so far contains only variables in their original form, i.e., without transformations such as normalization or standardization. We save it as `node_features_unnormalized.pkl` and leverage it later in the influencer determination.
In order to be able to use these node features for node classification and link prediction as well, we have to modify them a bit. First, we select only those node features relevant for the two tasks. For node classification as well as for link prediction, we have to transform the node features appropriately. Categorical node features are dummy-coded. For metric node features, there are two possibilities: On the one hand, they can be standardized to have mean 0 and variance 1. For this, the condition should be fulfilled that the metric node features are normally distributed or approximately normally distributed. On the other hand, they can be normalized so that they only assume values between 0 and 1. There are no special requirements for this. Figure 16 in appendix C shows how the individual metric node features are distributed. It can be clearly seen that none of the metric node features is normally distributed or approximately normally distributed. So, it follows that normalization must be used to transform the metric node features. The resulting node features dataset after

[28] The centrality measures were computed using networkx (Hagberg et al. (2008)). For an explanation of the individual centrality measures see appendix A.

dummy encoding and normalization is finally saved as `node_features.pkl`.

4.3.3 Edge features

We create an edge features dataset for each of our 5821 edges that includes the following three variables: "paper_link", "journal_link", and "hospital_link". Each of these three variables is 1 if the edge type in question exists between the respective two HCPs, and 0 otherwise. We save this dataset as `edge_features.pkl`. As mentioned above, we do not use these edge features in our later tasks. However, they are still useful as they record which edge types are hidden behind an edge in our HCP graph.

4.3.4 Summary of generated graph

To conclude this subsection, Figure 7 provides a visualization of our HCP graph, followed by a brief summary of its most important properties in Table 3.

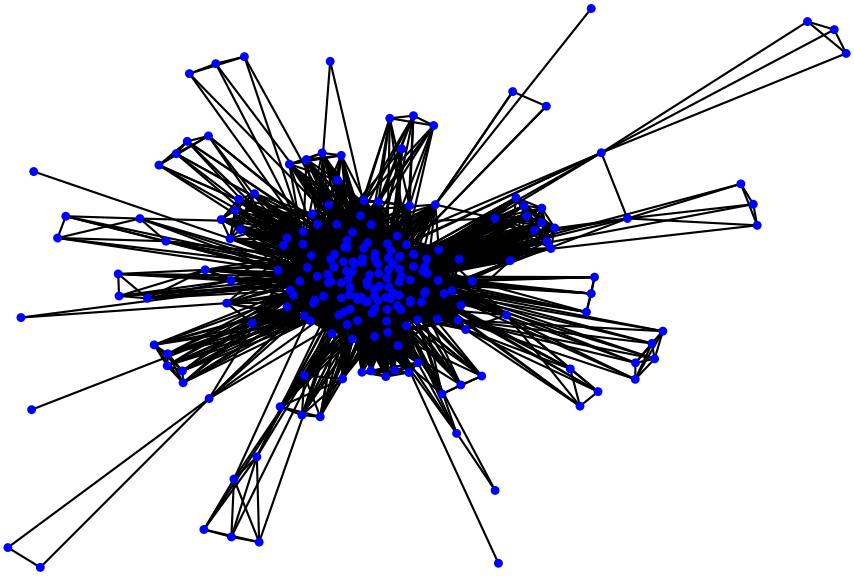


Figure 7: Generated HCP graph

On the whole, the HCP graph comes with a total of 229 nodes and 5821 links as well as 68 node features and 3 edge features. Labels do not exist yet. However, parts of the node features can become labels, e.g., in node classification. Apart from that, the HCP graph is both undirected and unweighted and has neither isolated nodes nor self-loops. It is striking that the node degrees have a wide span. The HCP with the fewest edges has only one edge,

Graph property	Value
Nodes	229
Edges	5821
Node features	68
Edge features	3
Labels	No
Directed?	No
Weighted?	No
Isolated nodes	0
Self-loops	0
Density	0.2230
Clustering coefficient	0.8055
Min degree	1
Max degree	127
Avg degree	50.8384
Degree assortativity	0.2047
Number of connected components	1
Size of largest component	229 (100.00%)

Table 3: Summary of generated HCP graph

while the one with the most edges has 127 edges. The entire graph is connected, so there is only one connected component.

Building on the node features and our HCP graph, we can finally start working on the four tasks of this project.

4.4 First task: influencer determination

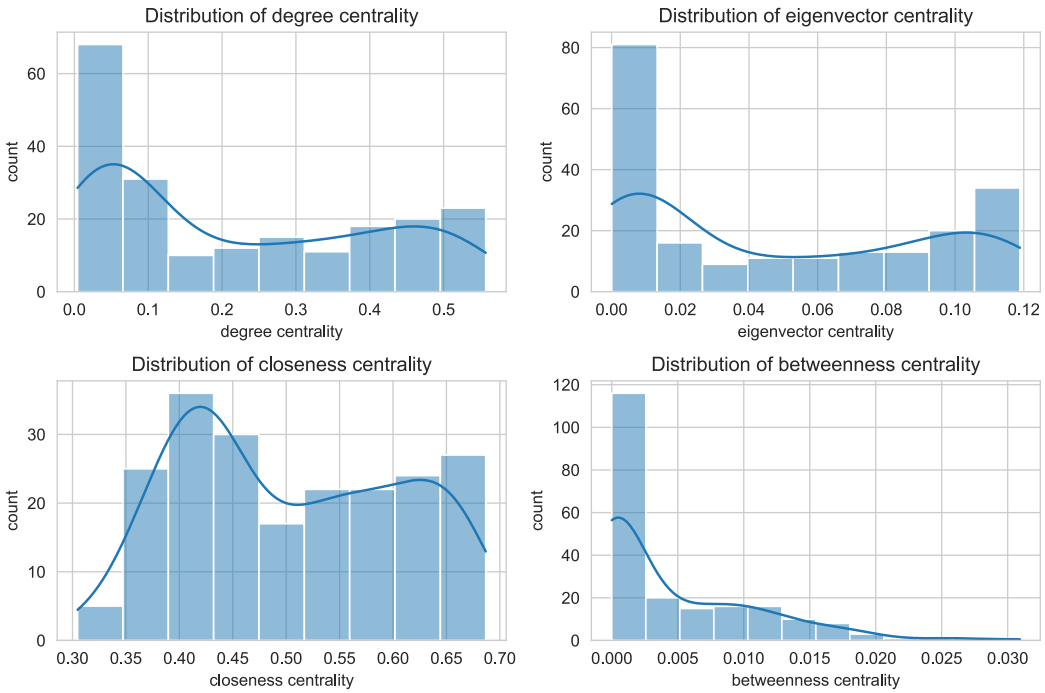
Within the scope of our first task, we assign an influencer status to the HCPs in the HCP graph generated above by considering their respective importance in the graph. Based on this, we aim to identify the top 5 influencers for the BC and melanoma indications.
We already know that 21 HCPs appear twice in the graph: first as BC HCP and second as melanoma HCP. In our influencer determination, we do not consider whether an HCP is a BC HCP or a melanoma HCP, but only evaluate an HCP's role in the whole graph. Therefore, we determine the influencer status only for physically distinct HCPs, so that instead of the previous 229 HCPs, we are left with only 208 physically distinct HCPs for the influencer determination after accounting for the 21 HCPs occurring twice.
The notebook `influencer_determination.ipynb` contains the code needed for this task.

As a starting point for our task, we use the dataset `node_features_unnormalized.pkl`,

which contains important unstandardized data on HCPs. This includes in particular the four centralities[29]: *degree centrality*, *eigenvector centrality*, *closeness centrality* and *betweenness centrality*[30]. Centrality values are computed based on the generated graph, which in turn is generated largely based on the scraped data, in which the indication that an HCP is responsible for is not considered. Thus, if an HCP is operating as both BC and melanoma HCP, their respective centralities across the two indications are the same.

4.4.1 Distribution of centrality measures

To get a better impression of the four centralities, we visualize their respective distributions in Figure 8. Table 4 provides their main location and dispersion measures.



These histograms show the respective distributions of the centralities, where the blue lines are kernel density estimations (kde), which are supposed to smooth the distributions and thus make them more recognizable.

Figure 8: Distributions of centralities

Overall, it can be seen that the value ranges of the centralities differ considerably between the four centralities. While the smallest values of degree centrality, eigenvector centrality and betweenness centrality are 0 or very close to 0, the smallest value of closeness centrality is

[29] An explanation of each centrality is provided in the glossary given in appendix A.

[30] It should be noted that we do not consider *PageRank*, another common centrality measure, here. This is because PageRank is designed for directed graphs (Currie Sivek (2021)).

	Degree	Eigenvector	Closeness	Betweenness
Mean	0.218518	0.048712	0.504291	0.004629
Standard deviation	0.185913	0.043429	0.103534	0.006137
Minimum	0.004386	0.000071	0.304813	0.000000
25% quantile	0.043860	0.005934	0.416058	0.000000
50% quantile	0.135965	0.034168	0.487179	0.001133
75% quantile	0.399123	0.094627	0.598425	0.008228
Maximum	0.557018	0.118865	0.686747	0.030924

Table 4: Summary of centralities

slightly above 0.3. Furthermore, degree centrality and closeness centrality with values above 0.5 and slightly below 0.7, respectively, can have significantly higher values than eigenvector centrality with values slightly below 0.12 and especially betweenness centrality with values slightly above 0.03.

We observe that the distributions of degree centrality and eigenvector centrality are similar. Both have a high number of HCPs in the very lower region of the distribution near the value 0. There are significantly fewer HCPs in the middle region of the distribution for both distributions, while there are again significantly more HCPs in the upper region of the distribution for both centralities, but noticeably fewer than in the very lower region.

The distribution of closeness centrality differs from the distributions of degree centrality and eigenvector centrality in that there are very few HCPs in the very lower region, while most HCPs are in the lower middle and upper middle regions. However, as with degree centrality and eigenvector centrality, there are significantly more HCPs in the upper region.

The distribution of betweenness centrality is much more different from the distributions of closeness and eigenvector centrality. More than half of the HCPs are in the very lower region of the distribution, while the rest of the HCPs are relatively evenly distributed across the rest of the distribution, with fewer and fewer HCPs having higher and higher values.

4.4.2 Assignment of influencer status

In the next step, we appropriately combine the centralities explored above and assign an influencer status to each HCP on this basis.

To consider all four centralities equally when assigning an influencer status and account for their different value ranges, we propose the following approach[31]: We assign a rank for each of the four centralties to each of the 208 HCPs, resulting in each HCP obtaining

[31] This approach is subjective and based on our assessment of how the various centrality measures can be meaningfully combined.

a "degree centrality rank", "eigenvector centrality rank", "closeness centrality rank" and "betweenness centrality rank". The motive is that a higher value of a centrality indicates a greater importance in the respective context of the centrality. Thus, for each of the four centralities, a rank is assigned to each HCP, where - for each centrality - the HCP with the highest value of this centrality receives rank 1, while the HCP with the lowest value of this centrality receives rank 208. Should several HCPs have the same value for a centrality, then the average rank is assigned to all of them for this centrality. After each HCP has been assigned a rank for each of the four centralities, the average of the four ranks is calculated for each HCP and thus the final rank is determined for each HCP, which is called "mean rank".

Based on the mean rank, an influencer status is assigned to each HCP according to one of 6 different groups, which correspond to those suggested by Sanders (2022). Table 5 shows which influencer status is assigned to which respective quantile of the mean rank.

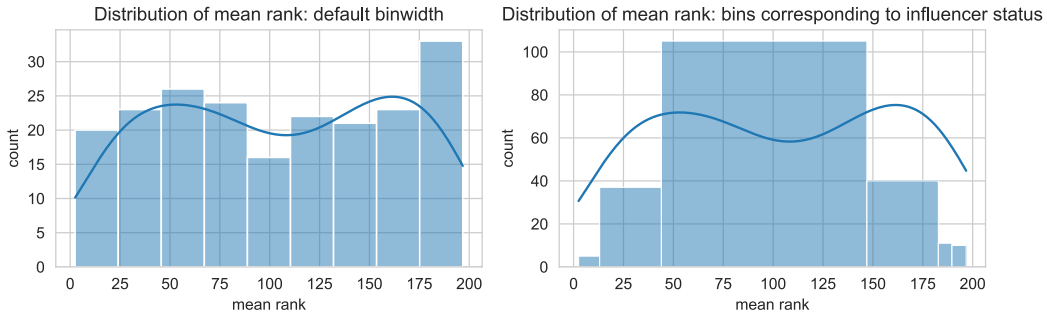
Influencer status	Mean rank score
Mega influencer	top 2%
Macro influencer	top 20%
Mid-tier influencer	top 70%
Micro influencer	top 90%
Nano influencer	top 95%
No influencer	below top 95%

Note that the influencer status is assigned without overlap: The score "top 20%" means that all HCPs whose mean rank is below the top 2% but within the top 20% are assigned "macro influencer" as influencer status.

Table 5: Influencer status by mean rank score

After calculating the mean rank and assigning an influencer status, we now take a deeper look at the mean rank. Figure 9 visualizes the distribution of mean rank and Table 6 summarizes its main location and dispersion measures. Note that here a lower mean rank of an HCP means that they are more important in the graph and have a higher influencer status.

In the left figure, a distribution can be observed for the mean rank that is much more similar to a uniform distribution than is the case with the distributions of the centralities. This is evident from the fact that the bars generally have a similar height and the kernel density estimation does not show such strong fluctuations as in the case of the distributions of the centralities. However, in the kernel density estimation we recognize two peaks around 50 and 160. A dip is present at about 110.



The histogram on the left-hand side displays the distribution of mean rank with binwidth being set to its default value. In contrast, the histogram on the right-hand side displays the distribution of mean rank using the bins that define the different categories of influencer status from Table 5.

Figure 9: Distribution of mean rank

	Mean rank
Mean	104.50000
Standard deviation	57.99863
Minimum	2.50000
25% quantile	52.00000
50% quantile	106.25000
75% quantile	155.87500
Maximum	196.75000

Table 6: Summary of mean rank

The right figure reflects our assignment of HCPs to influencer groups. The mid-tier influencer group is by far the largest, followed by the macro influencer group and the micro influencer group, which are roughly the same size. The mega influencer group, nano influencer group and no influencer group are by far the smallest.

4.4.3 Analysis of influencer groups

Likewise, we can gain a deeper insight into the 6 influencer groups and provide some summary statistics. Table 7 shows the actual quantiles that result from the theoretical quantiles in Table 5 after the HCPs have been divided into influencer groups, and gives the true percentage of HCPs in each influencer group. Table 8 shows how the total number in each influencer group from Table 7 is distributed across the values of the two important variables "account_type" and "indication".

We can see that the quantiles in "mean rank score" used to classify HCPs into influencer groups differ slightly from the resulting quantiles in "true mean rank score". The reason for

Influencer status	Mean rank score	True mean rank score	Total
Mega influencer	top 2%	top 2.4038%	5
Macro influencer	top 20%	top 20.1923%	37
Mid-tier influencer	top 70%	top 70.6731%	105
Micro influencer	top 90%	top 89.9038%	40
Nano influencer	top 95%	top 95.1923%	11
No influencer	below top 95%	below top 95.1923%	10

Table 7: Influencer groups: totals

Influencer status	**Total**	**Account type**		**Indication**	
		Doctor	Nurse	BC	Melanoma
Mega influencer	5	4	1	5	0
Macro influencer	37	35	2	31	6
Mid-tier influencer	105	75	30	67	38
Micro influencer	40	22	18	24	16
Nano influencer	11	6	5	10	1
No influencer	10	3	7	10	0

Table 8: Influencer groups: counts of important variables

this is that 208 is not divisible by 100 without a remainder.

Of the 208 HCPs, 145 are doctors and 63 are nurses, while 147 are BC HCPs and 61 are melanoma HCPs.

With regard to the variable "account_type", it is noticeable that the ratio of doctors to nurses is particularly high among the mega influencers and the macro influencers compared with the overall ratio of doctors and nurses. Among the mid-tier influencers, doctors and nurses are represented in roughly the same proportion as overall. Among micro influencers, nano influencers and no influencers, the ratio of doctors to nurses is small compared to the overall ratio of doctors and nurses. In these three groups, nurses are therefore over-represented. Overall, it can be seen that doctors dominate mega influencers and macro influencers, while nurses dominate micro influencers, nano influencers and no influencers. Among the mid-tier influencers, both are represented in roughly equal proportions. This finding is not surprising: Doctors are much more involved in research and therefore usually have a much larger network than nurses, which makes them significantly more influential.[32] Concerning the variable "indication", we observe that the ratio of melanoma HCPs to BC

[32]Note that nurses in Sweden must have completed a bachelor's degree in nursing and therefore have an academic background unlike in other countries.

HCPs is much smaller than the overall ratio of melanoma HCPs to BC HCPs for both mega influencers and macro influencers and for nano influencers and no influencers. The mega influencers and no influencers do not include a single melanoma HCP. However, among mid-tier influencers and micro influencers, melanoma HCPs are overrepresented compared with the overall ratio of melanoma HCPs to BC HCPs. Overall, therefore, the particularly influential and the particularly weakly influential HCPs belong almost exclusively to the BC HCPs, while the melanoma HCPs dominate the mid-tier and micro influencers in proportion. The reason that melanoma HCPs do not occur at all among mega influencers and only rarely among macro influencers could be that among our 208 different HCPs there are less than half as many melanoma HCPs as BC HCPs. Because melanoma HCPs are naturally more likely to form connections with other melanoma HCPs in the graph than with BC HCPs, the number of possible connections for melanoma HCPs is significantly smaller than for BC HCPs, which tends to make it more difficult for them than for BC HCPs to take an influential role in the graph.

4.4.4 Recommendation of top influencers

Finally, as each individual HCP has been assigned a mean rank and an associated influencer status, we can now also determine the 5 most important BC influencers and the 5 most important melanoma influencers. Our objective is to give Novartis an indication of which influencers they should work with more closely to increase sales of both their BC drug Kisqali and their melanoma drug combination Tafinlar-Mekinist.

The 5 most important BC influencers are given by the 5 BC HCPs with the smallest mean rank. Similarly, the 5 most important melanoma influencers are the 5 melanoma HCPs with the smallest mean rank. Table 9 contains the top 5 BC influencers, while Table 10 contains the top 5 melanoma influencers. Both tables also report the mean rank and influencer status for each HCP.

Name	Mean rank	Influencer status
1. Per Karlsson	2.5000	Mega influencer
2. Roger Henriksson	3.7500	Mega influencer
3. Tommy Fornander	7.1250	Mega influencer
4. Per Malmström	11.2500	Mega influencer
5. Yvonne Wengström	12.8750	Mega influencer

Table 9: Top 5 BC influencers

We see from the two tables above that the top 5 BC influencers are also the 5 most influential

Name	Mean rank	Influencer status
1. Gustav Ullenhag	14.2500	Macro influencer
2. Hildur Helgadottir	16.1250	Macro influencer
3. Lars Ny	20.3750	Macro influencer
4. Max Levin	29.3750	Macro influencer
5. Karin Johansson	36.8750	Macro influencer

Table 10: Top 5 melanoma influencers

HCPs in the entire HCP graph. All of them are mega influencers. The top 5 melanoma influencers, on the other hand, are macro influencers. These results are consistent with the data from Table 8 on the distribution of BC HCPs and melanoma HCPs across influencer groups. We also note that of these 10 HCPs, none is both a BC HCP and a melanoma HCP. Novartis should specifically target these 10 HCPs because they are outstandingly influential and therefore promise to boost sales of Novartis drugs.
Not only are these 10 HCPs well-connected, as indicated by their degree centrality, but they also have a broad network through their one-hop neighbors, as implied by their eigenvector centrality. Furthermore, their importance is also due to the fact that they often lie on the shortest paths between two nodes in the graph, as shown by their closeness centrality, and also serve as bridges between nodes, which is expressed by their betweenness centrality.

Having completed the first of our four tasks, we have become more familiar with our data, so that we can move on to the next three tasks where the different GNNs come to the fore. When working with machine learning models and particularly GNNs, two Python libraries are immensely useful: PyTorch (Paszke et al. (2019)) and PyTorch Geometric (Fey and Lenssen (2019)). While PyTorch generally offers the possibility to implement neural networks relatively easily, PyTorch Geometric is based on PyTorch and is specifically designed for GNNs. Especially for node classification and link prediction, PyTorch Geometric offers the opportunity to use a large number of pre-programmed GNN architectures, thus reducing programming effort. Therefore, PyTorch Geometric is our preferred library for node classification and link prediction. Concerning time series forecasting, however, PyTorch Geometric does not provide an implementation of the GTS model. Instead, we implement it using PyTorch, which implies significantly higher programming effort.

4.5 Second task: node classification

In the context of node classification our goal is to predict for each HCP, i.e., for each node in our HCP graph, the segment that they are associated with. An HCP always belongs to one of the following three segments.

- *S1*: prescriber or non-prescriber who influences decision making on patient treatment within this market at a regional or national level.
- *S2*: prescriber or non-prescriber who is responsible for decision making on the patient treatment for this market within their own account.
- *S3*: clinician or nurse, responsible for initiating medical treatment.
- *S4*: clinician, nurse or non-prescriber, responsible for supporting diagnosis, treatment or management of the patient through their treatment journey.

In summary, the higher the segment to which an HCP belongs, the more influential they are. S1 is the highest segment and S4 the lowest.
The reason we choose the segment variable for node classification is that we suspect that including the graph structure in the predictions of the segment leads to a significant increase in predictive power. For example, we find that HCPs belonging to S1 have an average degree of 103.5625, while HCPs belonging to S2, S3 and S4 have average degrees of 62.4130, 51.3214 and 24.5806, respectively. We hope to be able to leverage such information contained in the graph structure to improve our predictions.
Table 11 shows the distribution of the segment variable.

Segment	Number of HCPs
S1	93
S2	66
S3	54
S4	16

Table 11: Distribution of segment

We see that all four segments are represented sufficiently often. Thus, there is no considerable class imbalance in the variable, so that we can use it for node classification.
Here, the datasets `node_features.pkl` and `edge_features.pkl` as well as the edges in `all_links.txt` serve as the data for node classification. These are further processed in the notebook `dataset_creation.ipynb`[33] to create the PyTorch Geometric dataset called

[33] For more information on dataset creation beyond this section, consult this notebook.

`NodeClassificationDataset`, which is usable for both our baselines and our GNNs. Specifically, we do the following: From `node_features.pkl` we create the labels, i.e., the segment variable, and node features[34]. The edges from `all_links.txt` are used to create the HCP graph and `edge_features.pkl` is used to assign the three edge features to each edge.[35]

For our applications, be it baselines or GNNs, we need to split the nodes into a training set, a validation set and a test set. The validation set is meant to enable us to validate the respective model's performance on the test set. The performance metric we use is the accuracy (ACC). Regarding the split, we would like to follow the approach of Kipf and Welling (2016) and therefore only have a small number of labeled nodes. In the datasets used in Kipf and Welling (2016), this corresponds to label rates of at most 5.2%. However, these datasets also have a very large number of nodes, the smallest being 2,708. We, on the other hand, only have a total of 229 nodes in our graph, which is considerably smaller. In order to have a sufficiently high number of nodes in the training set to enable our models to generalize to unlabeled nodes, we choose to have 40 nodes in the training set, 10 of which for each segment, thereby avoiding class imbalance. For the remaining 189 nodes, we aim to have a rough 1:2 split between the validation and test set. We therefore opt for 60 nodes in the validation set and for the remaining 129 nodes in the test set.[36]

In the node classification task, our procedure is as follows. First, we fit four baselines. These are the following standard machine learning methods for classification tasks: a logistic regression, a random forest classifier, a support vector classifier, and a multilayer perceptron, the latter already being a neural network. These baselines should allow us to evaluate the performance of the GNNs fit later in this task. More precisely, we see if the GNNs that can incorporate the graph structure into the forecasts in addition to the node features succeed in outperforming the baselines that can only consider the node features. After this, we fit the three GNNs introduced in section 3.2 for node classification tasks: a graph convolutional network, a GNN with Chebyshev convolution and a graph attention network. At the end, we briefly summarize our results and determine which method performed best.

Note that the losses and accuracies that result from a single train-validation-test split are still relatively strongly subject to chance. Therefore, we fit all our models on 10 different train-validation-test splits and average the obtained accuracy values at the end. Of course, for each of the 10 runs, the training, validation and test sets have the sizes described above.

[34] See appendix B for a table of these node features.

[35] Note that the edge features are not used in this task. They are only present in the dataset and may be included in future models.

[36] The standard benchmark dataset for semi-supervised node classification is Cora, which can also be found in PyTorch Geometric as part of the Planetoid dataset. It has a training set of size 140, a validation set of size 500 and a test set of size 1000. We are guided by this 1:2 split between the validation and test set.

We make our results replicable by setting a different seed for each of the 10 runs.[37] All metric values reported below are rounded to four decimal places. The best hyperparameter combination is identified as the hyperparameter combination that leads to the highest mean validation accuracy respectively the highest mean best validation accuracy.[38]
For each method, we document exactly how we apply it, which hyperparameters we tune, which hyperparameter combination performs best and what the overall best node classification performance is. Additionally, we also report the development of losses and accuracies for neural networks.

We commence with the application of our baselines, which include logistic regression, a random forest classifier, a support vector classifier and a multilayer perceptron as standard machine learning methods.

4.5.1 Logistic regression

Logistic regression is a method for predicting a discrete dependent variable based on a set of independent variables.[39] Like linear regression, it produces predictions based on a linear combination of the independent variables, but then transforms them using the sigmoid function. As a result, the predictions for the values of the dependent variable that logistic regression outputs lie between 0 and 1 and can be interpreted as probabilities that the dependent variable takes on this respective value. The value of the dependent variable that is assigned the highest probability is then taken as the final prediction (Pant (2019); Cancino (2021)).
The notebook `logistic_regression_baseline.ipynb` contains our entire implementation of logistic regression. We use the scikit-learn (Pedregosa et al. (2011)) implementation of logistic regression, where the main hyperparameters to tune are:

- *penalty*: specifies the norm of the penalty. The options are: 'none', where no penalty

[37] The 10 different seeds are given by 12345, 23456, 34567, 45678, 56789, 54321, 65432, 76543, 87654 and 98765.

[38] For each run, the logistic regression, the random forest classifier and the support vector classifier output only one accuracy for the training, validation and test set. Therefore, these 10 accuracies are averaged to obtain the mean training accuracy, mean validation accuracy and mean test accuracy, respectively. For the neural networks, i.e., the multilayer perceptron, graph convolutional network, GNN with Chebyshev convolution and graph attention network, we obtain the accuracies over all epochs of the training process. For each run, we always take the best accuracy for the training, validation and test set and calculate the mean best training accuracy, mean best validation accuracy, and mean best test accuracy over all 10 runs.

[39] Strictly speaking, logistic regression distinguishes between two cases. In the first case the dependent variable is binary, i.e., can take only two values. Then the method is also called binary logistic regression. In the second case, the dependent variable can take more than two values and is no longer referred to as binary, but generally as discrete. This is the more general of the two cases. The method is then referred to as multinomial logistic regression.

is added, 'l2', where an L2 penalty is added and which is the default choice, 'l1', where an L1 penalty is added, and 'elasticnet', where both L1 and L2 penalty terms are added. Here, we use all four possible penalties for hyperparameter tuning.

- C: inverse of regularization strength, where smaller values specify stronger regularization. The default value is 1.0. Here, we use the values 0.001, 0.01, 0.1, 1, 10, 100 and 1000 for hyperparameter tuning.

- *solver*: algorithm to use in the optimization problem. The options are: 'newton-cg', 'lbfgs', 'liblinear', 'sag' and 'saga'. The default solver is 'lbfgs'. With multiclass problems, only 'netwon-cg', 'lbfgs', 'sag' and 'saga' can handle multinomial loss, whereas 'liblinear' is limited to one-versus-rest schemes. Since we have a multiclass problem, we use only 'netwon-cg', 'lbfgs', 'sag' and 'saga' for hyperparameter tuning.

It should be noted that not all penalties can be combined with all solvers. Only the following combinations are possible: 'newton-cg' with 'l2' and 'none', 'lbfgs' with 'l2' and 'none', 'liblinear' with 'l1' and 'l2', 'sag' with 'l2' and 'none'. On the other hand, 'saga' can be combined with all solvers, i.e., 'elasticnet', 'l1', 'l2' and 'none'.
Table 12 summarizes all hyperparameters to be tuned with their respective values considered.

Hyperparameter	Values considered
penalty	'none', 'l2', 'l1', 'elasticnet'
C	0.001, 0.01, 0.1, 1, 10, 100, 1000
solver	'netwon-cg', 'lbfgs', 'sag', 'saga'
Total combinations: 70	

Table 12: Logistic regression: hyperparameters

In total, after subtracting the hyperparameter combinations for which penalty and solver cannot be combined, there are 70 hyperparameter combinations for which we fit the logistic regression to our data with 10 different runs each.
For each hyperparameter combination, the results from the 10 runs are saved in the dataset `lrc_results.pkl`. It contains the accuracies from the 10 runs for the training, validation and test set for each hyperparameter combination and also the respective means and standard deviations over the 10 runs for the training, validation and test set.
The best hyperparameter combination, i.e., the one that leads to the highest mean validation accuracy of 0.8517 over all 10 runs, is given in Table 13.

Hyperparameter	Best value
penalty	'l1'
C	10
solver	'saga'

Table 13: Logistic regression: best hyperparameter combination

In the following, Table 14 shows the means and standard deviations of the accuracies over the 10 runs for the training, validation and test set for the best hyperparameter combination.

	ACC
Training	0.9975 ± 0.0075
Validation	0.8517 ± 0.0486
Test	0.8775 ± 0.0312

Table 14: Logistic regression: mean ACCs with standard deviations

We can see that the accuracies in the training set are the best with a mean of 0.9975 and also have the lowest standard deviation with a value of 0.0075. Thus, the segment can be correctly predicted in the training set for almost all HCPs. The mean accuracy in the validation set is 0.8517, while the mean accuracy in the test set is 0.8775. The two mean accuracies are close to each other and thus the performance on the validation set confirms that the performance on the test set is valid. Overall, the mean accuracies in the validation and test set are noticeably lower than in the training set and the standard deviations of the accuracies are also significantly higher at 0.0486 and 0.0312. This may be due to the fact that the training set is the smallest in terms of size and that we generalize from such a small set to so many other HCPs.

4.5.2 Random forest classifier

A random forest is a meta estimator that fits a number of decision tree classifiers on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting (Pedregosa et al. (2011)).
The notebook `random_forest_classifier_baseline.ipynb` contains our entire implementation of the random forest classifier. We use the scikit-learn (Pedregosa et al. (2011)) implementation of a random forest classifier, where the main hyperparameters to tune are:

- *n_estimators*: the number of trees in the forest. Here, we use the values 100 and 500 for hyperparameter tuning.

- *max_depth*: the maximum depth of the tree. We use the values 1, 2, 3 and 4 for hyperparameter tuning.
- *min_samples_split*: the minimum number of samples required to split an internal node. The values 2, 5 and 10 are considered for hyperparameter tuning.
- *min_samples_leaf*: the minimum number of samples required to be at a leaf node. Hyperparameter tuning is performed with the values 1, 2 and 4.
- *max_features*: the number of features to consider when looking for the best split. If set to 'auto', the square root of the number of features is used. If 'log2' is used, then the logarithm to base 2 of the number of features is used. If None, then the entire number of features is used. Here, we consider the values 2, 4, 'log2', equal to 6, 'auto', equal to 8, 10, and None, equal to 64, for hyperparameter tuning.
- *bootstrap*: whether bootstrap samples are used when building trees. If False, the whole dataset is used to build each tree. For hyperparameter tuning, we consider both values True and False.

Table 15 summarizes all hyperparameters to be tuned with their respective values considered.

Hyperparameter	Values considered
n_estimators	100, 500
max_depth	1, 2, 3, 4
min_samples_split	2, 5, 10
min_samples_leaf	1, 2, 4
max_features	2, 4, 'log2', 'auto', 10, None
bootstrap	True, False
Total combinations: 864	

Table 15: Random forest classifier: hyperparameters

All in all, there are 864 hyperparameter combinations for which we fit the random forest classifier to our data with 10 different runs each.
For each hyperparameter combination, the results from the 10 runs are saved in the dataset `rfc_results.pkl`. It contains the accuracies from the 10 runs for the training, validation and test set for each hyperparameter combination and also the respective means and standard deviations over the 10 runs for the training, validation and test set.
There are two hyperparameter combinations that lead to the highest mean validation accuracy of 0.8450. However, one of the two hyperparameter combinations shows higher values

for the mean training accuracy and the mean test accuracy. Thus, this is the best hyperparameter combination. It is given is Table 16.

Hyperparameter	Best value
n_estimators	100
max_depth	4
min_samples_split	2
min_samples_leaf	1
max_features	None
bootstrap	False

Table 16: Random forest classifier: best hyperparameter combination

Table 17 shows the means and standard deviations of the accuracies over the 10 runs for the training, validation and test set for the best hyperparameter combination.

	ACC
Training	0.9800 ± 0.0245
Validation	0.8450 ± 0.0558
Test	0.8558 ± 0.0418

Table 17: Random forest classifier: mean ACCs with standard deviations

We can observe that the accuracies in the training set are the best with a mean value of 0.9800 and also have the lowest standard deviation with a value of 0.0245. Thus, the segment can be correctly predicted in the training set for almost all HCPs. The mean accuracy in the validation set is 0.8450, while the mean accuracy in the test set is 0.8558. The two mean accuracies are similar to each other so that the performance on the validation set confirms the performance on the test. Overall, the mean accuracies in the validation and test set are noticeably lower than in the training set and the standard deviations of the accuracies are also significantly higher at 0.0558 and 0.0418.

To get an impression of which features have a particular impact on the predictions, we can look at the feature importances for our random forest classifier.
We now determine the feature importances in all 10 runs of our best hyperparameter combination. These are saved in `rfc_feature_importances.pkl`. Figure 17 in appendix D visualizes the feature importances depending on the feature index and Table 81 in appendix D shows which feature is assigned to which feature index.
We can see that the feature importances are very similar across the 10 runs. A lot of

feature importances are zero or extremely low so that they are not visible in the plots anymore. The feature importances are by far highest for features 22 (Doctor), 23 (Nurse) and for features 61 (T1), 62 (T2), 63 (T3). These features belong to the original variables "account_type" and "tier". The features 16 (num_papers_norm) as well as 17 (degree_norm), 18 (degree_centrality_norm), 19 (eigenvector_centrality_norm), 20 (closeness_centrality_norm) and 21 (betweenness_centrality_norm) have much lower but still moderately high values in nearly all runs.
We can therefore conclude that only few features are important. By a wide margin, the variables "account_type" and "tier" are most important. This is a sensible finding since doctors are more likely found in higher segments, while nurses are rather found in lower segments. In addition, "tier" is correlated with "segment", the correlation being 0.2468, and it is therefore sensible to observe high feature importances for "tier". Variables such as the number of papers published and the graph connectivity measures such as degree and other centrality measures are of secondary importance. It is still reasonable that these variables are more important than the rest of variables because a high number of papers indicates a higher influencer status and therefore a higher segment. In a similar fashion, better connectivity suggests a greater network, thereby a higher influencer status and thereby a higher segment. The rest of variables is virtually unimportant for the prediction of segment.

4.5.3 Support vector classifier

The objective of a support vector classifier is to find a hyperplane in an N-dimensional space, N being the number of features, that distinctly classifies the data points (Gandhi (2018)).
The notebook `support_vector_classifier_baseline.ipynb` includes our entire implementation of the support vector classifier. We use the scikit-learn (Pedregosa et al. (2011)) implementation of a support vector classifier, so that we obtain the main hyperparameters to tune as:

- C: regularization parameter, where the strength of the regularization is inversely proportional to C. It must be strictly positive and is a squared L2 penalty. We consider the values 0.001, 0.01, 0.1, 1, 10, 100 and 1000 for hyperparameter tuning.
- *kernel*: specifies the kernel type to be used in the algorithm. The options are: 'linear', 'poly', 'rbf', 'sigmoid' and 'precomputed'. If none is given, 'rbf' will be used. For hyperparameter tuning, the kernels 'linear', 'poly', 'rbf' and 'sigmoid' are used, whereas 'precomputed' is not used.

- *degree*: degree of the polynomial kernel function 'poly'. It is ignored by all other kernels. Here, we take the values 0, 1, 2, 3, 4, 5 and 6 into account for hyperparameter tuning.

- *gamma*: kernel coefficient for 'rbf', 'poly' and 'sigmoid'. If 'scale', which is the default, is passed, then $\frac{1}{\text{n\_features} \cdot \text{X.var()}}$ is used as value of gamma. If 'auto', then $\frac{1}{\text{n\_features}}$ is used as value of gamma. We consider 'scale', 'auto', 0.01, 0.1, 1, 10 and 100 as values for hyperparameter tuning.

Table 18 gives all hyperparameters with the respective values to consider for tuning.

Hyperparameter	Values considered
C	0.001, 0.01, 0.1, 1, 10, 100, 1000
kernel	'linear', 'poly', 'rbf', 'sigmoid'
degree	0, 1, 2, 3, 4, 5, 6
gamma	'scale', 'auto', 0.01, 0.1, 1, 10, 100
Total combinations: 1372	

Table 18: Support vector classifier: hyperparameters

Overall, there is a total of 1372 hyperparameter combinations for which we fit the support vector classifier to our data with 10 different runs each.
For each hyperparameter combination, the results from the 10 runs are saved in the dataset `svm_results.pkl`. It contains the accuracies from the 10 runs for the training, validation and test set for each hyperparameter combination and also the respective means and standard deviations over the 10 runs for the training, validation and test set.
There are two hyperparameter combinations that lead to the highest mean validation accuracy of 0.8767. They lead to exactly the same training accuracies, validation accuracies and test accuracies over the 10 runs, so that the means and standard deviations of the accuracies in the training, validation and test set are also the same. Table 19 provides the two best hyperparameter combinations.

Hyperparameter	Comb. 1: best value	Comb. 2: best value
C	0.1	100
kernel	'poly'	'poly'
degree	3	3
gamma	1	0.1

Table 19: Support vector classifier: best hyperparameter combinations

Table 20 shows the means and standard deviations of the accuracies over the 10 runs for the training, validation and test set, which are identical for the two best hyperparameter combinations.

	ACC
Training	0.9975 ± 0.0075
Validation	0.8767 ± 0.0374
Test	0.8922 ± 0.0215

Table 20: Support vector classifier: mean ACCs with standard deviations

We observe that the accuracies in the training set are the best with a mean value of 0.9975 and also have the lowest standard deviation with a value of 0.0075. Thus, the segment can be correctly predicted in the training set for almost all HCPs. The mean accuracy in the validation set is 0.8767, while the mean accuracy in the test set is 0.8922. Since the two mean accuracies are close to each other, the performance on the test set can be validated by the performance on the validation set. All in all, the mean accuracies in the validation and test set are noticeably lower than in the training set and the standard deviations of the accuracies are also significantly higher at 0.0374 and 0.0215.

4.5.4 Multilayer perceptron

Although the MLP is still a baseline, it nevertheless initiates an important transition: the transition to neural networks. Hence, the documentation of the MLP is clearly more similar to that of the GNNs than to the documentation of the previous three baselines. So, we also take a look here at the learned embeddings as well as the loss and accuracy curves across the different epochs, just as we do later for the three GNNs.

An MLP is a neural network where the mapping between inputs and output is non-linear. It has an input and an output layer, and one or more hidden layers with many neurons stacked together. Neurons in an MLP can use any arbitrary activation function. An MLP is a feedforward algorithm because inputs are combined with the initial weights in a weighted sum and subjected to the activation function. Each layer feeds the next one with the result of their computation. This continues all the way through the hidden layers to the output layer (Bento (2021)).

The notebook `multilayer_perceptron_baseline.ipynb` contains our entire implementation of the MLP. With the PyTorch (Paszke et al. (2019)) implementation of a linear

operator, we fit an MLP that has only one hidden layer in addition to the input layer and the output layer to obtain an architecture that prevents overfitting. This is particularly imperative due to our relatively small number of only 229 nodes.
The input dimension is 64, corresponding to the number of node features, while the output dimension is 4, corresponding to the number of segments. The hidden dimension, i.e., the dimension of the node embeddings in the hidden layer, can be tuned. We also apply dropout after the hidden layer, where the dropout probability can also be tuned.
We choose the cross entropy loss as the loss function and Adam as the optimizer, for which the learning rate and weight decay can be tuned.
To further counteract overfitting, we also apply early stopping, i.e., if the validation loss does not decrease after a predefined number of epochs, the patience, the training process is aborted.
In addition, for each run, the losses and accuracies for the best epoch, i.e., the epoch with the lowest validation loss, are saved.
The main hyperparameters to tune are:

- *hidden_channels*: dimension of the node embeddings in the hidden layer. The values considered for hyperparameter tuning are 8, 16 and 32.
- *dropout_prob*: probability of a node to be dropped. We consider the values 0.1, 0.3, 0.5 and 0.7 for hyperparameter tuning.
- *learning_rate*: learning rate of the Adam optimizer. Here, we use 0.001, 0.01 and 0.1 for hyperparameter tuning.
- *weight_decay*: L2 penalty added to the loss function to prevent overfitting. For hyperparameter tuning, we take the values 5e-1, 5e-2, 5e-3, 5e-4, 5e-5 and 5e-6 into account.
- *patience*: number of epochs after which to stop the training process if no decrease in the validation loss has been observed in the meantime. We take the values 3, 5 and 10 into account for hyperparameter tuning.

Table 21 summarizes all hyperparameters to be tuned with their respective values considered. In total, there are 648 hyperparameter combinations for which we fit the MLP to our data with 10 different runs each.
For each hyperparameter combination, the results from the 10 runs are saved in the dataset

Hyperparameter	Values considered
hidden_channels	8, 16, 32
dropout_prob	0.1, 0.3, 0.5, 0.7
learning_rate	0.001, 0.01, 0.1
weight_decay	5e-1, 5e-2, 5e-3, 5e-4, 5e-5, 5e-6
patience	3, 5, 10
Total combinations: 648	

Table 21: Multilayer perceptron: hyperparameters

`mlp_results.pkl`. It contains the best accuracies from the 10 runs for the training, validation and test set for each hyperparameter combination and also the respective means and standard deviations over the 10 runs for the training, validation and test set.
There are two hyperparameter combinations that lead to the highest mean best validation accuracy of 0.8633. They lead to exactly the same best training accuracies, best validation accuracies and best test accuracies over the 10 runs, so that the means and standard deviations of the best accuracies in the training, validation and test set are also the same. Table 22 provides the two best hyperparameter combinations.

Hyperparameter	Comb. 1: best value	Comb. 2: best value
hidden_channels	32	32
dropout_prob	0.1	0.1
learning_rate	0.1	0.1
weight_decay	0.05	0.05
patience	5	10

Table 22: Multilayer perceptron: best hyperparameter combinations

Table 23 shows the means and standard deviations of the best accuracies over the 10 runs for the training, validation and test set for the two best hyperparameter combinations.

	Best ACC
Training	0.9850 ± 0.0166
Validation	0.8633 ± 0.0452
Test	0.8806 ± 0.0228

Table 23: Multilayer perceptron: mean best ACCs with standard deviations

It is evident that the best accuracies in the training set are the highest with a mean value of 0.9850 and also have the lowest standard deviation with a value of 0.0166. Thus, the

segment can be correctly predicted in the training set for almost all HCPs. The mean best accuracy in the validation set is 0.8633, while the mean best accuracy in the test set is 0.8806. The two mean best accuracies are close to each other and thus the performance on the validation set confirms that the performance on the test set is valid. Overall, the mean best accuracies in the validation and test set are noticeably lower than in the training set and the standard deviations of the best accuracies are also significantly higher at 0.0452 and 0.0228.

Overall, we have an MLP with the following transformations

```
MLP(
    (lin1): Linear(in_features=64, out_features=32, bias=True)
    (lin2): Linear(in_features=32, out_features=4, bias=True)
).
```

Table 24 shows how the total parameters of the MLP are distributed among its individual transformations.

Layer.Parameter	Parameter tensor shape	Number of parameters
lin1.weight	[32, 64]	2048
lin1.bias	[32]	32
lin2.weight	[4, 32]	128
lin2.bias	[4]	4
Total params: 2212		
Trainable params: 2212		
Non-trainable params: 0		

Table 24: MLP model parameters

Of the total 2212 parameters, the vast majority of $(64 + 1) \cdot 32 = 2080$ parameters belong to the first linear operator, the remaining $(32 + 1) \cdot 4 = 132$ parameters to the second linear operator.

To better understand the training process, we now take a look at how the losses and accuracies develop over the epochs. The corresponding figures can be found in appendix D.

Figure 18 and Figure 19 show how the losses in the training, validation and test set develop

over the epochs for the two best hyperparameter combinations.
The losses from Figure 18 are a subset of the losses from Figure 19 for each run, because the training process in Figure 19 with a patience of 10 was always carried out 5 epochs further for each run than in Figure 18 with a patience of 5. Otherwise, all other hyperparameters are the same. Therefore, we refer exclusively to Figure 19 in the description here.
First of all, it can be seen that the validation loss, which serves as an abort criterion for early stopping, no longer decreases in any run in the last 5 epochs. Therefore, the results for the means and standard deviations of the best accuracies are exactly the same for the two hyperparameter combinations.
We see that the training process stops relatively quickly for each run and that the three losses show a similar pattern over the 10 runs. From epoch 1 to 8, the three losses decrease constantly and then reach a value from which they tend to move slightly upwards again with smaller fluctuations.
The training loss is by far the smallest in all runs from epoch 6 onward. The validation loss and test loss are relatively similar for the 10 runs, but can also show larger differences. As a rule, the validation loss is somewhat larger than the test loss.

Figure 20 and Figure 21 plot how the accuracies in the training, validation and test set develop over the epochs for the two best hyperparameter combinations.
Again, for the reason described above, Figure 20, which belongs to the hyperparameter combination with a patience of 5, is for each run a subset of Figure 21, which belongs to the hyperparameter combination with a patience of 10. Therefore, we refer exclusively to Figure 21.
At the very beginning of the training process, the validation accuracy and test accuracy - and for some runs also the training accuracy - drop relatively sharply, but then rise again immediately and reach a value around which they remain with smaller fluctuations. For all runs, the training accuracy is by far the highest from epoch 8 onwards with values close to 1, while the validation accuracy and the test accuracy usually hover around values between 0.8 and 0.9 from epoch 8 onwards. Generally speaking, the validation accuracy is lower than the test accuracy.

Finally, for each run we consider the best learned embeddings, i.e., the learned embeddings from the epoch in which the lowest validation loss was realized. Since our output layer is four-dimensional because we have four segments, our learned embeddings are also

four-dimensional. Of course, these embeddings cannot be plotted in this dimension. Therefore, in scikit-learn we use the *t-NSE* (VAN DER MAATEN AND HINTON (2008)), short for "t-distributed stochastic neighbor embedding", function to transform the learned four-dimensional embeddings into two-dimensional embeddings. The two dimensions of the resulting embeddings do not have interpretability. The transformation should only enable the plotting of the embeddings in order to find out whether four different classes - as in our variable "segment" - can also be discovered in the embeddings.
The figures showing the learned embeddings for all 10 runs can be found in appendix D as Figure 22 and Figure 23. Note that the figures do not only contain the learned embeddings for the nodes in the training set, but also for the nodes in the validation and test set. It is immediately noticeable that the two figures are identical, which - as explained above - is due to the fact that the training processes of the two hyperparameters combinations differ only by the last 5 epochs, in which, however, the epoch in which the lowest validation loss was realized is not included.
Therefore, we refer only to one of the two figures, here Figure 23. The legend below the figure shows which color belongs to which segment.
Overall, we strongly see the formation of embedding clusters, i.e., of different groups of HCPs for which very similar embeddings were learned. This makes sense in light of the different segments. Clearly, four clusters can be identified for run 2, 3, 4, 5, 7, and 8. Sometimes, the transitions between different clusters are rather smoother, as for run 1, 6, 10. For run 9, a total of three clusters can be seen.
However, we also recognize that despite the clustering, HCPs of different segments are still represented in the same cluster. This means that if there are e.g. four clusters, each cluster does not exclusively contain HCPs of one segment, but - as we can see - also HCPs of other segments, although potentially to a lesser extent.

After implementing all four baselines, we proceed with the application of the GNNs, which include the graph convolutional network, the GNN with Chebyshev convolution, and the graph attention network. The documentation of these GNNs is very similar to the documentation of the MLP above.

4.5.5 Graph convolutional network

We continue with the GCN already introduced in section 3.2.1. The entire implementation of the GCN can be found in notebook `graph_convolutional_network.ipynb`. With the

PyTorch Geometric (Fey and Lenssen (2019)) implementation of a convolutional layer, we fit a GCN that has only one hidden layer in addition to the input layer and the output layer. The underlying motive here, as with the MLP, is that we want to obtain an architecture that counteracts overfitting, especially because of our relatively small number of 229 nodes. As with the MLP, the input dimension is 64 according to the number of node features and the output dimension is 4 according to the number of segments. The hidden dimension can be tuned, as can the dropout probability, which is used when applying dropout after the hidden layer.
Again, we use the cross entropy loss here and choose Adam as the optimizer. The learning rate and weight decay can also be tuned here.
We also apply early stopping again to reduce overfitting and to stop the training process if the validation loss does not decrease after a predefined number of epochs.
For each run, the losses and accuracies are saved for the best epoch, i.e., the epoch with the lowest validation loss.
The main hyperparameters to tune are:

- *hidden_channels*: dimension of the node embeddings in the hidden layer. The values considered for hyperparameter tuning are 8, 16 and 32.
- *dropout_prob*: probability of a node to be dropped. We consider the values 0.01, 0.05, 0.1, 0.3, 0.5 and 0.7 for hyperparameter tuning.
- *learning_rate*: learning rate of the Adam optimizer. Here, we use 0.001, 0.01 and 0.1 for hyperparameter tuning.
- *weight_decay*: L2 penalty added to the loss function to prevent overfitting. For hyperparameter tuning, we take the values 5e-1, 5e-2, 5e-3, 5e-4, 5e-5 and 5e-6 into account.
- *patience*: number of epochs after which to stop the training process if no decrease in the validation loss has been observed in the meantime. We take the values 3, 5 and 10 into account for hyperparameter tuning.

Table 25 summarizes all hyperparameters to be tuned with their respective values considered. In total, there are 972 hyperparameter combinations for which we fit the GCN to our data with 10 different runs each.
For each hyperparameter combination, the results from the 10 runs are saved in the dataset

Hyperparameter	Values considered
hidden_channels	8, 16, 32
dropout_prob	0.01, 0.05, 0.1, 0.3, 0.5, 0.7
learning_rate	0.001, 0.01, 0.1
weight_decay	5e-1, 5e-2, 5e-3, 5e-4, 5e-5, 5e-6
patience	3, 5, 10
Total combinations: 972	

Table 25: Graph convolutional network: hyperparameters

`gcn_results.pkl`. It contains the best accuracies from the 10 runs for the training, validation and test set for each hyperparameter combination and also the respective means and standard deviations over the 10 runs for the training, validation and test set.
The hyperparameter combination that leads to the highest mean best validation accuracy of 0.4883 is given in Table 26.

Hyperparameter	Best value
hidden_channels	16
dropout_prob	0.5
learning_rate	0.1
weight_decay	0.0005
patience	5

Table 26: Graph convolutional network: best hyperparameter combination

Table 27 shows the means and standard deviations of the best accuracies over the 10 runs for the training, validation and test set for the best hyperparameter combination.

	Best ACC
Training	0.4000 ± 0.0894
Validation	0.4883 ± 0.0460
Test	0.4612 ± 0.0589

Table 27: Graph convolutional network: mean best ACCs with standard deviations

Two key differences to the results of the previous models are immediately apparent here: Firstly, the mean best accuracies in the GCN are much lower and secondly, the mean best accuracies in the validation and test set are higher than in the training set.
Overall, the best accuracies in the validation set are still the highest with a mean value of 0.4883 and have the lowest standard deviation with a value of 0.0460. The best accuracies in

the test set with a mean value of 0.4612 and a standard deviation of 0.0589 are similar and thus the performance on the test set is confirmed. In the training set, the best accuracies are significantly lower with a mean value of 0.4 and also have the highest standard deviation with a value of 0.0894. In particular, it should be emphasized that the performance on the training set is even slightly below the performance of random guessing, i.e., the prediction of the most frequently occurring segment for all HCPs, which would result in an accuracy of 0.4061.[40]

Altogether, we obtain a GCN with the following transformations

```
GCN(
    (conv1): GCNConv(64, 16)
    (conv2): GCNConv(16, 4)
).
```

Table 28 shows how the total parameters of the GCN are distributed among its individual transformations.

Layer.Parameter	Parameter tensor shape	Number of parameters
conv1.bias	[16]	16
conv1.lin.weight	[16, 64]	1024
conv2.bias	[4]	4
conv2.lin.weight	[4, 16]	64
Total params: 1108		
Trainable params: 1108		
Non-trainable params: 0		

Table 28: GCN model parameters

Of the total 1108 parameters, the vast majority of $(64+1)\cdot 16 = 1040$ parameters are associated with the first graph convolutional operator, the remaining $(16+1)\cdot 4 = 68$ parameters with the second graph convolutional operator.

So as to better understand the training process, we now consider how the losses and accuracies develop over the epochs. The corresponding figures can be found in appendix D.

[40] This value results as the share of the number of HCPs in the most frequently occurring segment in the total number of HCPs. As Table 11 shows, this proportion is $\frac{93}{229} = 0.4061$.

Figure 24 plots how the losses in the training, validation and test set develop over the epochs for the best hyperparameter combination.
It can be seen that, as with the MLP, the training process stops relatively quickly for each run. The validation and test losses are very similar for each run, but show different patterns between the runs. However, they fluctuate strongly in each run and tend to remain constant or even increase slightly.
The training loss, on the other hand, shows significantly smaller fluctuations for each run and is the only one of the three losses to decrease.

In Figure 25, we see how the accuracies in the training, validation and test set develop over the epochs for the best hyperparameter combination.
The validation and test accuracies are relatively similar for the 10 runs, but sometimes also show larger differences. They both fluctuate strongly and tend to remain constant.
In contrast, the training accuracy is increasing despite larger fluctuations for the 10 runs.

Eventually, for each run we look at the best learned embeddings, i.e., the learned embeddings from the epoch in which the lowest validation loss was observed. As with the MLP, we use the t-NSE (van der Maaten and Hinton (2008)) function to convert the learned four-dimensional embeddings into two-dimensional embeddings and plot them. Again, our goal is to find out whether four classes - as in our variable "segment" - can be discovered in the embeddings.
Figure 26 visualizes the learned embeddings for all 10 runs. Again, note that the figure does not only contain the learned embeddings for the nodes in the training set, but also for the nodes in the validation and test set.
It can be observed that for all runs the embeddings - sometimes with larger and sometimes with hardly any deviations - appear to be arranged in a straight line. Embedding clusters are generally not visible, only in run 2, 5, 6 and 7 small clusters away from the embeddings which appear to be in a straight line can be detected. In run 3 and 9 the embeddings are more scattered.
Overall, no meaningful embeddings seem to have been learned by the GCN, with the help of which the HCPs could be correctly assigned to the different segments.

4.5.6 GNN with Chebyshev convolution

Having applied our GCN, we now turn to the GNN with Chebyshev convolution, where the convolutional filter in the Fourier domain used in the GCN is approximated with a truncated expression of Chebyshev polynomials up to K^{th} order as in formulation (3.9) .

The notebook `gnn_with_chebyshev_convolution.ipynb` contains our entire implementation of the GNN with Chebyshev convolution. With the PyTorch Geometric (Fey and Lenssen (2019)) implementation of a Chebyshev spectral graph convolutional operator, we fit a GNN with Chebyshev convolution that has only one hidden layer in addition to the input layer and the output layer. The reason here, as with the MLP and GCN, is that we want to have an architecture that helps prevent overfitting, especially because of our relatively small number of 229 nodes.

As before, the input dimension is 64 according to the number of node features and the output dimension is 4 according to the number of segments. The hidden dimension can be tuned as well as the dropout probability, which is used when applying dropout after the hidden layer.

The model performance also depends on the order K up to which we consider Chebyshev polynomials. Therefore, tuning K is crucial.

Again, we use the cross entropy loss here and choose Adam as the optimizer. The learning rate and weight decay can also be tuned here.

We also apply early stopping again to reduce overfitting and to stop the training process if the validation loss does not decrease after a predefined number of epochs.

For each run, the losses and accuracies are saved for the best epoch, i.e., the epoch with the lowest validation loss.

The main hyperparameters to tune are:

- *hidden_channels*: dimension of the node embeddings in the hidden layer. The values considered for hyperparameter tuning are 8, 16 and 32.
- K: Chebyshev filter size. This gives us the highest order of the Chebyshev polynomials. The values 1 and 2 are used for hyperparameter tuning in order to achieve some regularization.
- *dropout_prob*: probability of a node to be dropped. We consider the values 0.1, 0.3 and 0.5 for hyperparameter tuning.
- *learning_rate*: learning rate of the Adam optimizer. Here, we use 0.001, 0.01 and 0.1 for hyperparameter tuning.

- *weight_decay*: L2 penalty added to the loss function to prevent overfitting. For hyperparameter tuning, we take the values 5e-1, 5e-2, 5e-3 and 5e-4 into account.
- *patience*: number of epochs after which to stop the training process if no decrease in the validation loss has been observed in the meantime. We take the values 5 and 10 into account for hyperparameter tuning.

Table 29 provides a summary of all hyperparameters to be tuned with their respective values considered.

Hyperparameter	Values considered
hidden_channels	8, 16, 32
K	1, 2
dropout_prob	0.1, 0.3, 0.5
learning_rate	0.001, 0.01, 0.1
weight_decay	5e-1, 5e-2, 5e-3, 5e-4
patience	5, 10
Total combinations: 432	

Table 29: GNN with Chebyshev convolution: hyperparameters

In total, there are 432 hyperparameter combinations for which we fit the GNN with Chebyshev convolution to our data with 10 different runs each.
For each hyperparameter combination, the results from the 10 runs are saved in the dataset `cheb_results.pkl`. It contains the best accuracies from the 10 runs for the training, validation and test set for each hyperparameter combination and also the respective means and standard deviations over the 10 runs for the training, validation and test set.
The hyperparameter combination that leads to the highest mean best validation accuracy of 0.8583 is given in Table 30. There are two hyperparameter combinations that lead to the highest mean validation accuracy of 0.8583. However, one of the two hyperparameter combinations shows higher values for the mean best training accuracy and the mean best test accuracy. Thus, this is the best hyperparameter combination. It is given is Table 30.
Table 31 shows the means and standard deviations of the best accuracies over the 10 runs for the training, validation and test set for the best hyperparameter combination.
It indicates that the best accuracies in the training set are the highest with a mean value of 0.9800 and also have the lowest standard deviation with a value of 0.0150. Thus, the segment can be correctly predicted in the training set for almost all HCPs. The mean best accuracy in the validation set is 0.8583, while the mean best accuracy in the test set is 0.8760. The two mean best accuracies are close to each other and thus the performance on

Hyperparameter	Best value
hidden_channels	32
K	1
dropout_prob	0.1
learning_rate	0.1
weight_decay	0.05
patience	10

Table 30: GNN with Chebyshev convolution: best hyperparameter combination

	Best ACC
Training	0.9800 ± 0.0150
Validation	0.8583 ± 0.0344
Test	0.8760 ± 0.0183

Table 31: GNN with Chebyshev convolution: mean best ACCs with standard deviations

the validation set confirms that the performance on the test set is valid. Overall, the mean best accuracies in the validation and test set are noticeably lower than in the training set and the standard deviations of the best accuracies are also significantly higher at 0.0344 and 0.0183. Overall, these results bear a strong resemblance to those of the baselines and the MLP.

All in all, we have a GNN with Chebyshev convolution with the following transformations

```
ChebNet(
    (conv1): ChebConv(64, 32, K=1, normalization=sym)
    (conv2): ChebConv(32, 4, K=1, normalization=sym)
).
```

Table 32 shows how the total parameters of the GNN with Chebyshev convolution are distributed among its individual transformations.
Of the total 2212 parameters, the vast majority of $(64+1)\cdot 32 = 2080$ parameters belong to the first Chebyshev spectral graph convolutional operator, the remaining $(32+1)\cdot 4 = 132$ parameters to the second Chebyshev spectral graph convolutional operator.

In order to better understand the training process, we now take a look at how the losses and

Layer.Parameter	Parameter tensor shape	Number of parameters
conv1.bias	[32]	32
conv1.lins.0.weight	[32, 64]	2048
conv2.bias	[4]	4
conv2.lins.0.weight	[4, 32]	128
Total params: 2212		
Trainable params: 2212		
Non-trainable params: 0		

Table 32: GNN with Chebyshev convolution model parameters

accuracies develop over the epochs. The corresponding figures can be found in appendix D.

Figure 27 shows how the losses in the training, validation and test set develop over the epochs for the best hyperparameter combination.
We see that the training process stops relatively quickly for each run and that the three losses show a similar pattern over the 10 runs. From epoch 1 to 8, the three losses decrease constantly and then reach a value from which they tend to move slightly upwards again with smaller fluctuations.
The training loss is by far the smallest in all runs from epoch 6 onward. The validation loss and test loss are relatively similar for the 10 runs, but can also show larger differences. Generally, the validation loss is somewhat larger than the test loss.

Figure 28 shows how the accuracies in the training, validation and test set develop over the epochs for the best hyperparameter combination.
At the very beginning of the training process, the validation accuracy and test accuracy - and for some runs also the training accuracy - drop relatively lightly, but then rise again and reach a value around which they remain with smaller fluctuations. For all runs, the training accuracy is by far the highest from epoch 8 onwards with values close to 1, while the validation accuracy and the test accuracy usually hover around values between 0.8 and 0.9 from epoch 8 onwards. As a rule, the validation accuracy is lower than the test accuracy.

Finally, for each run we consider the best learned embeddings, i.e., the learned embeddings from the epoch in which the lowest validation loss was observed. As before, we use the t-NSE (van der Maaten and Hinton (2008)) function to convert the learned four-dimensional embeddings into two-dimensional embeddings and plot them. Again, our goal is to find out

whether the four classes - as in our variable "segment" - can be discovered in the embeddings.
Figure 29 plots the learned embeddings for all 10 runs. Again, note that the figure does not only contain the learned embeddings for the nodes in the training set, but also for the nodes in the validation and test set.
We can observe that, similar to the MLP, embedding clusters also emerge here when learning the embeddings. However, four clusters, which would make sense against the background of the four segments, are only clearly recognizable for run 7, 8 and 10. For run 1, 4, 5 and 9, we recognize 3 clusters. For run 2 and 3, the transitions between clusters are more fluid and therefore it is unclear how many embedding clusters there are. A very strong scattering of embeddings prevails for run 6.
However, we also recognize that despite the clustering, HCPs of different segments are still represented in a cluster. This means that if there are e.g. four clusters, each cluster does not exclusively contain HCPs of one segment, but - as we can see - also HCPs of other segments, even though possibly to a lesser degree.

In overall terms, the results we obtain for the GNN with Chebyshev convolution are similar to those of the MLP. There are large similarities in the losses and accuracies. The learned embeddings of the MLP, on the other hand, are somewhat easier to assign to different clusters than is the case for the learned embeddings of the GNN with Chebyshev convolution.

4.5.7 Graph attention network

We now conclude with the application of the GAT, which automatically learns the importance of each neighbor based on the attention mechanism.
In the notebook `graph_attention_network.ipynb`, we provide our entire implementation of the GAT. Using the PyTorch Geometric implementation of a graph attentional operator (Fey and Lenssen (2019)), we fit a GAT that has only one hidden layer in addition to the input layer and the output layer. The background here, as with the MLP and the preceding GNNs, is that we want to have an architecture that helps prevent overfitting, especially because of our relatively small number of 229 nodes.
As before, the input dimension is 64 according to the number of node features and the output dimension is 4 according to the number of segments. The hidden dimension can be tuned, just as the dropout probability, which is used when applying dropout after the input layer and after the hidden layer.

The model performance also depends on the number of multi-head-attentions. Therefore, tuning this hyperparameter is also important.
Again, we use the cross entropy loss here and choose Adam as the optimizer. The learning rate and weight decay can also be tuned here.
We also apply early stopping again to reduce overfitting and to stop the training process if the validation loss does not decrease after a predefined number of epochs.
For each run, the losses and accuracies are saved for the best epoch, i.e., the epoch with the lowest validation loss.
The main hyperparameters to tune are:

- *hidden_channels*: dimension of the node embeddings in the hidden layer. The values considered for hyperparameter tuning are 8, 16 and 32.
- *heads*: number of multi-head-attentions. The values 1 and 2 are used for hyperparameter tuning.
- *dropout_prob*: probability of a node to be dropped. We consider the values 0.1, 0.3 and 0.5 for hyperparameter tuning.
- *learning_rate*: learning rate of the Adam optimizer. Here, we use 0.001, 0.01 and 0.1 for hyperparameter tuning.
- *weight_decay*: L2 penalty added to the loss function to prevent overfitting. For hyperparameter tuning, we take the values 5e-1, 5e-2, 5e-3 and 5e-4 into account.
- *patience*: number of epochs after which to stop the training process if no decrease in the validation loss has been observed in the meantime. We take the values 5 and 10 into account for hyperparameter tuning.

For a summary of all hyperparameters with their respective values considered for tuning, take a look at Table 33.
In sum, we have are 432 hyperparameter combinations for which the GAT is fit to our data with 10 different runs each.
For each hyperparameter combination, the results from the 10 runs are saved in the dataset `gat_results.pkl`. It contains the best accuracies from the 10 runs for the training, validation and test set for each hyperparameter combination and also the respective means and standard deviations over the 10 runs for the training, validation and test set.
The hyperparameter combination that leads to the highest mean best validation accuracy of 0.4600 is given in Table 34.

Hyperparameter	Values considered
hidden_channels	8, 16, 32
heads	1, 2
dropout_prob	0.1, 0.3, 0.5
learning_rate	0.001, 0.01, 0.1
weight_decay	5e-1, 5e-2, 5e-3, 5e-4
patience	5, 10
Total combinations: 432	

Table 33: Graph attention network: hyperparameters

Hyperparameter	Best value
hidden_channels	8
heads	1
dropout_prob	0.3
learning_rate	0.1
weight_decay	0.0005
patience	10

Table 34: Graph attention network: best hyperparameter combination

Table 35 shows the means and standard deviations of the best accuracies over the 10 runs for the training, validation and test set for the best hyperparameter combination.

	Best ACC
Training	0.3950 ± 0.0678
Validation	0.4600 ± 0.0821
Test	0.4357 ± 0.0814

Table 35: Graph attention network: mean best ACCs with standard deviations

We observe here the same two phenomena that we have already encountered in the GCN: first, that the mean best accuracies in the GAT are significantly lower than in the baselines and in the GNN with Chebyshev convolution, and second, that the mean best accuracies in the validation and test set are higher than in the training set.
All in all, the best accuracies in the validation set are still the highest with a mean value of 0.4600 but also have the highest standard deviation with a value of 0.0821. The best accuracies in the test set with a mean value of 0.4357 and a standard deviation of 0.0814 are similar, which corroborates that the performance on the test set is valid. In the training set, the best accuracies are significantly lower with a mean value of 0.3950 but also have the

lowest standard deviation with a value of 0.0678. As with the GCN, we emphasize that the performance on the training set is even slightly below the performance of random guessing, which - as we saw above - would result in an accuracy of 0.4061.

Overall, we get a GAT with the following transformations

```
GAT(
    (conv1): GATConv(64, 8, heads=1)
    (conv2): GATConv(8, 4, heads=1)
).
```

Table 36 shows how the total parameters of the GAT are distributed among its individual transformations.

Layer.Parameter	Parameter tensor shape	Number of parameters
conv1.att_src	[1, 1, 8]	8
conv1.att_dst	[1, 1, 8]	8
conv1.bias	[8]	8
conv1.lin_src.weight	[8, 64]	512
conv2.att_src	[1, 1, 4]	4
conv2.att_dst	[1, 1, 4]	4
conv2.bias	[4]	4
conv2.lin_src.weight	[4, 8]	32
Total params: 580		
Trainable params: 580		
Non-trainable params: 0		

Table 36: GAT model parameters

Of the total 580 parameters, the vast majority of 536 parameters belong to the first graph attentional operator, the remaining 44 parameters to the second graph attentional operator.

In order to better understand the training process, we now take a look at how the losses and accuracies develop over the epochs. The corresponding figures can be found in appendix D.

Figure 30 shows how the losses in the training, validation and test set develop over the epochs for the best hyperparameter combination.
It can be seen that, as with the GCN, the training process stops relatively quickly for each run, though not as quickly as with the MLP and the GNN with Chebyshev convolution. The

validation and test losses are very similar for each run, but show different patterns between the runs. However, they fluctuate strongly in each run and tend to remain constant or even increase slightly.
The training loss, on the other hand, shows significantly smaller fluctuations for each run and is the only one of the three losses to decrease.

Figure 31 shows how the accuracies in the training, validation and test set develop over the epochs for the best hyperparameter combination.
The validation and test accuracies are relatively similar for the 10 runs and only rarely show larger differences. They both fluctuate strongly and tend to remain constant.
In contrast, the training accuracy for the 10 runs is increasing and shows larger fluctuations.

Finally, for each run we consider the best learned embeddings, i.e., the learned embeddings from the epoch in which the lowest validation loss was observed. As before, we use the t-NSE (van der Maaten and Hinton (2008)) function to convert the learned four-dimensional embeddings into two-dimensional embeddings and plot them. Again, our goal is to find out whether the four classes - as in our variable "segment" - can be discovered in the embeddings. Figure 32 visualizes the learned embeddings for all 10 runs. Again, note that the figure does not only contain the learned embeddings for the nodes in the training set, but also for the nodes in the validation and test set.
As with the GCN, it can be seen that for most runs the embeddings - sometimes with larger and sometimes with hardly any deviations - appear to be arranged in a straight line. This is particularly well observed for runs 4, 7, 8, 9 and 10. Embedding clusters are generally not visible. However, in run 3, 4, 5, 7 and 9 small clusters away from the embeddings which appear to be in a straight line can be detected. In run 1, 2, 3 and 6 the embeddings are more scattered.
Overall, no meaningful embeddings seem to have been learned by the GAT, with the help of which the HCPs could be correctly assigned to the different segments.

4.5.8 Summary and comparison

To conclude our node classification task, we briefly summarize and compare the performance of all methods - baselines and GNNs - on the test set.

Table 37 shows the means and standard deviations of the accuracies in the test set realized

for the baselines in the 10 runs. These are visualized by the boxplots in Figure 10.

Method	ACC
Logistic regression	0.8775 ± 0.0312
Random forest classifier	0.8558 ± 0.0418
Support vector classifier	0.8922 ± 0.0215
Multilayer perceptron	0.8806 ± 0.0228

Table 37: Node classification: results of baselines

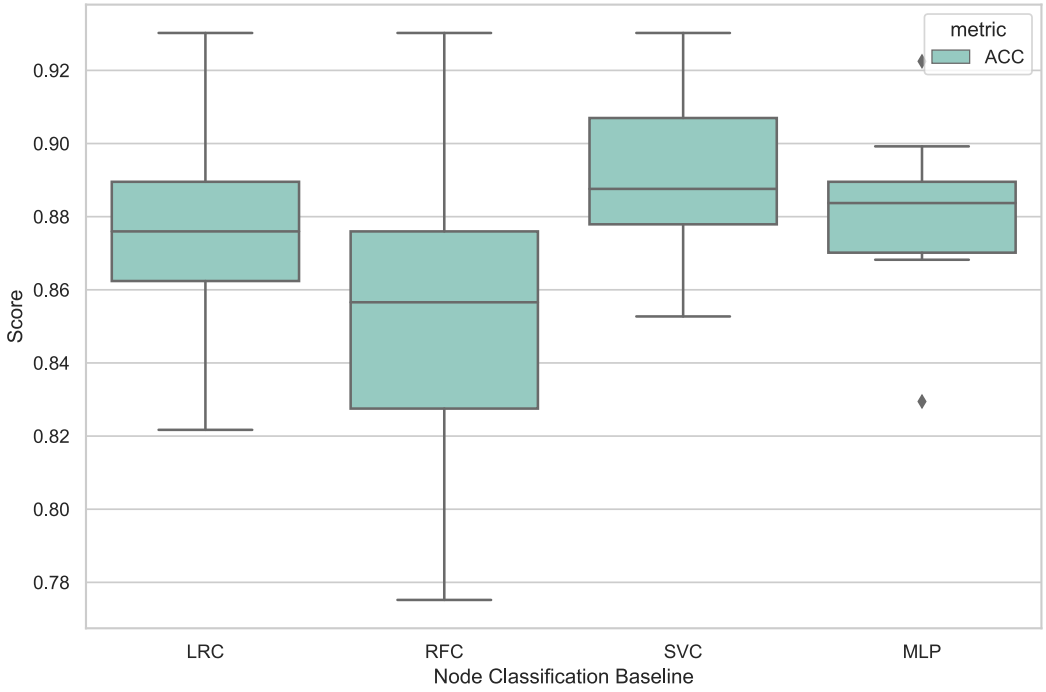


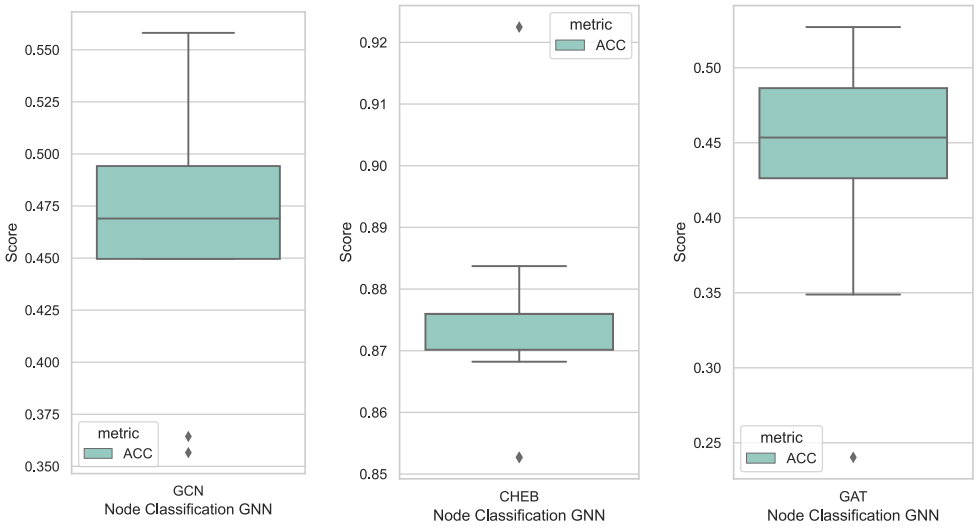
Figure 10: Node classification: boxplots of baseline results

It is evident that the support vector classifier demonstrates the best performance on the test set because its mean test accuracy is the highest. In addition, its standard deviation of the test accuracies over the 10 runs is also the lowest of all four baselines, so that the support vector classifier also offers the most reliable performance. However, it should be noted that the other three baselines are only slightly behind and thus also prove to perform well.

In a similar fashion, Table 38 gives us the means and standard deviations of the best accuracies in the test set realized for the GNNs for the 10 runs. The boxplots in Figure 11 plot these.

Method	Best ACC
Graph convolutional network	0.4612 ± 0.0589
GNN with Chebyshev convolution	0.8760 ± 0.0183
Graph attention network	0.4357 ± 0.0814

Table 38: Node classification: results of GNNs



Note that we plot the best test accuracies here in three boxplots rather than one boxplot, because the best test accuracies are so different in magnitude across the three GNNs that it would be difficult to discern anything in a single boxplot.

Figure 11: Node classification: boxplots of GNN results

We note that among the GNNs, the GNN with Chebyshev convolution performs the best - both in terms of the mean best test accuracy and the standard deviation of the best accuracies in the test set for the 10 runs. The other two GNNs, the GCN and the GAT, are far behind and have not only very low mean best accuracies in the test set, but also more strongly fluctuating best accuracies in the test set for the 10 runs.

In conclusion, for our node classification task at large, the support vector classifier is the best performing model due to its highest mean test accuracy. The baselines generally show a similarly good performance. Among the GNNs, only the GCN with Chebyshev convolution can keep up with this performance. Yet, it has the smallest standard deviation of the best accuracies in the test set for the 10 runs and its results are thus the most reliable. The GCN and GAT fall short of showing their potential in this task.

4.6 Third task: link prediction

In our link prediction task, we aim to predict whether there is a link between any two nodes in the HCP graph. So this means that we want to predict whether any two HCPs are linked. In section 4.3, the HCP graph was constructed in such a way that two HCPs are linked if they have published at least one paper together, have published in at least one journal together, or work at the same hospital.

The reason we perform this link prediction task is that we think that jointly considering the graph structure and node features can lead to increased link prediction power. It is conceivable that two HCPs are more likely to be linked if they share common features, such as operating for the same indication or belonging to the same tier. However, this cannot be accounted for by the link prediction baselines.

Unlike in node classification, where we used the same PyTorch Geometric dataset for the baselines and the GNNs, here we do not feed the baselines and the GNNs with exactly the same dataset. In this link prediction task, we work with two different data sources. We only pass the baselines the entire set of edges contained in `all_links.txt`. The use of a PyTorch Geometric dataset is not necessary here, since we mainly only work with functions and implementations from scikit-learn (Paszke et al. (2019)), networkx (Hagberg et al. (2008)) and networkit (Staudt et al. (2016)). For our applications involving GNNs, however, we create a PyTorch Geometric dataset called `LinkPredictionDataset`, because the GNNs are implemented with Pytorch and PyTorch Geometric, so working with a PyTorch Geometric dataset is particularly suitable. From `node_features.pkl`, we create the node features, whereas the edges from `all_links.txt` are used to create the HCP graph. To assign the three features to each edge, we use `edge_features.pkl`.[41]

Note that for both baselines and GNNs, in addition to the positive training, validation and test edges, i.e., the edges that actually exist in the HCP graph, we also use negative training, validation and test edges, i.e., edges that do not exist in the HCP graph. We always use as many negative edges as positive edges for the training, validation and test set in order to balance the learning between positive and negative edges. The reason why negative edges are also used is that we want to determine not only how reliably the presence of a positive edge is predicted, but also how reliably the absence of a negative edge is predicted.

Overall, we use a 0.6-0.1-0.3 train-validation-test split, where the validation set again serves to confirm the performance on the test set. Table 39 shows how many of the 5821 positive

[41] The edge features are not used in this task. However, they may be included in additional models in the future.

and negative edges are in the training, validation and test set. Different from the node clas-

Type	Number of edges			Sum
	Training	Validation	Test	
Positive	3493	582	1746	5821
Negative	3493	582	1746	5821

Table 39: Number of positive and negative edges in training, validation and test set

sification task, where there were very few samples, i.e., nodes, for the training process, there is now a sufficient number of samples, i.e., edges, for link prediction to ensure a meaningful analysis.

Note that the edges contained in the positive and negative training, validation and test edges can differ between baselines and GNNs. This is because the way that the positive and negative training, validation and test edges are sampled from the set of all edges is not the same for the baselines and the GNNs, which use the PyTorch Geometric dataset. However, the sets containing the positive and negative training, validation and test edges always have the same size.

All in all, we again perform 10 runs for each model - as in node classification - to ensure that the losses and especially the average precision (AP) and area under the curve (AUC) that we use as metric do not depend too much on chance. Therefore, we fit all our models to 10 different train-validation-test splits and average the obtained average precision and area under the curve values[42] at the end. Of course, the training, validation and test sets are as large as described above for each run. We make our results replicable by setting a different seed for each of the 10 runs.[43] All metric values reported below are rounded to four decimal places.

Performing these 10 runs also means that the results of the baselines and the GNNs, which are based on 10 runs with potentially different edges in the positive and negative training, validation and test sets, are comparable.

Our decision as to which model performs best is based on the highest mean validation AP respectively the highest mean best validation AP. For the best model in this respect, we report and compare the performance on the test set at the end.

[42] Technically speaking, computing the mean of the AUC is not entirely correct. That is why we do not use it to find the best hyperparameter combination. The mean AUC, however, should still give us an indication of the performance to expect over the 10 runs in terms of the AUC.

[43] The 10 different seeds are given by 12345, 23456, 34567, 45678, 56789, 54321, 65432, 76543, 87654 and 98765.

We choose the following procedure: First, we fit a rich set of baselines, including all 5 local heuristics presented, i.e., common neighbors, Jaccard score, preferential attachment, Adamic-Adar, resource allocation, one global heuristic, i.e., Katz index, spectral clustering, node2vec and DeepWalk as a special case of node2vec. These baselines are meant to allow us to evaluate the performance of the GNNs at the end of this task. More precisely, we can then verify if the GNNs that can incorporate the graph structure and the node features succeed in outperforming the baselines that can only consider the graph structure. Following this, we fit the 3 GNNs introduced in subsection 3.3 for link prediction tasks: the graph autoencoder, the variational graph autoencoder and SEAL. Finally, we briefly summarize our results and determine which method performs best.
For each method, we document exactly how it is applied, which hyperparameters are tuned, which hyperparameter combination performs best and what the overall best link prediction performance is. Besides, we also document the development of losses and metrics for the GNNs.

We start with the heuristics, the first part of our baselines, which include common neighbors, Jaccard score, preferential attachment, Adamic-Adar and resource allocation as local heuristics as well as Katz index as a global heuristic. Rooted PageRank and SimRank are not implemented by us.

For the six heuristics, we use a uniform procedure. We first load the entire set of edges from `all_links.txt`, create a networkx graph from them, and then perform the train-validation-test split using the created networkx graph, so that we obtain the adjacency matrix of the training graph in addition to the positive and negative training, validation, and test edges. Using the information about the positive and negative training edges, we create the training graph. Finally, we can obtain link predictions for all edges that are not part of the positive or negative training edges. This is done for Adamic-Adar, Jaccard score, preferential attachment and resource allocation using networkx (Hagberg et al. (2008)) functions, which output the respective score for any link that is not part of the training edges. For common neighbors there is no direct implementation, we determine the score using another function in networkx. The Katz index is also not implemented in networkx, but in networkit (Staudt et al. (2016)), another less well-known library for graphs. At the end, for each metric we divide all obtained scores by the largest score, so that all scores are between 0 and 1.

We perform 10 runs with the seeds given above. The size of the positive and negative training, validation, test edges is of course always unchanged.
Our metrics AP and AUC can only be calculated for the validation and test set. We always save our respective results in a file which contains the validation APs and AUCs, test APs and AUCs for the 10 runs, the mean validation AP and AUC, mean test AP and AUC over 10 runs, the corresponding standard deviations of the validation APs and AUCs as well as the corresponding standard deviations of the test APs and AUCs.

4.6.1 Jaccard score

The implementation can be found in the notebook `jaccard_baseline.ipynb` and is based on Hu (2018). The networkx function `jaccard_coefficient` provides a direct implementation of the Jaccard score and calculates the Jaccard score for all edges that are not part of the positive or negative training edges. There are no hyperparameters to tune. The results are saved in `jc_results.pkl`. Table 40 shows the mean AP and mean AUC over the 10 runs in the validation and test set with the corresponding standard deviations.

	AP	AUC
Validation	0.9537 ± 0.0067	0.9543 ± 0.0073
Test	0.9545 ± 0.0027	0.9545 ± 0.0027

Table 40: Jaccard score: mean APs and mean AUCs with standard deviations

4.6.2 Preferential attachment

The notebook `preferential_attachment_baseline.ipynb` contains our implementation of the heuristic, which is based on Hu (2018). We are provided with a direct implementation of preferential attachment through the networkx function `preferential_attachment`, which calculates the respective scores for all edges that are not part of the positive or negative training edges. There are no hyperparameters to tune. The results are saved in `pa_results.pkl`. Table 41 shows the mean AP and mean AUC over the 10 runs in the validation and test set with the corresponding standard deviations.

4.6.3 Adamic-Adar

In the notebook `adamic_adar_baseline.ipynb`, our implementation of the heuristic can be found, which is based on Hu (2018). The networkx function `adamic_adar_index` provides

	AP	AUC
Validation	0.9327 ± 0.0068	0.9185 ± 0.0085
Test	0.9306 ± 0.0037	0.9159 ± 0.0045

Table 41: Preferential attachment: mean APs and mean AUCs with standard deviations

a direct implementation of Adamic-Adar and calculates the respective scores for all edges that are not part of the positive or negative training edges. There are no hyperparameters to tune. We save the results as `aa_results.pkl`. Table 42 shows the mean AP and mean AUC over the 10 runs in the validation and test set with the corresponding standard deviations.

	AP	AUC
Validation	0.9576 ± 0.0062	0.9570 ± 0.0070
Test	0.9573 ± 0.0028	0.9567 ± 0.0034

Table 42: Adamic-Adar: mean APs and mean AUCs with standard deviations

4.6.4 Resource allocation

The notebook `resource_allocation_baseline.ipynb` includes our implementation of the heuristic, which builds on Hu (2018). The networkx function `resource_allocation_index` gives us a direct implementation of resource allocation and calculates the respective scores for all edges that are not part of the positive or negative training edges. There are no hyperparameters to tune. The results are saved in `ra_results.pkl`. Table 43 shows the mean AP and mean AUC over the 10 runs in the validation and test set with the corresponding standard deviations.

	AP	AUC
Validation	0.9712 ± 0.0059	0.9701 ± 0.0067
Test	0.9703 ± 0.0025	0.9692 ± 0.0031

Table 43: Resource allocation: mean APs and mean AUCs with standard deviations

4.6.5 Common neighbors

We implement the heuristic in the notebook `common_neighbors_baseline.ipynb`. There is no function in networkx that implements the common neighbors heuristic directly. Therefore

we use another function in networkx, `common_neighbors`, which gives us a list of common neighbors. We can then use the number of elements in this list as the common neighbors score. To achieve uniformity in the implementation of the heuristics, we compute the score for all edges that are not part of the positive or negative training edges. There are no hyperparameters to tune. The results are saved in `cn_results.pkl`. Table 44 shows the mean AP and mean AUC over the 10 runs in the validation and test set with the corresponding standard deviations.

	AP	AUC
Validation	0.9485 ± 0.0067	0.9501 ± 0.0075
Test	0.9479 ± 0.0031	0.9494 ± 0.0036

Table 44: Common neighbors: mean APs and mean AUCs with standard deviations

4.6.6 Katz index

The notebook `katz_baseline.ipynb` contains our implementation of the heuristic. There is no function in networkx that implements the Katz index. We therefore resort to another library that also specializes in graphs but is less common: networkit. We first convert the networkx graph into a networkit graph, and then we can use the function `KatzIndex` from networkit to compute the Katz index scores for edges that are not in the training graph, and then retrieve them using the `run` function. For the sake of uniformity in the implementation of the heuristics, we compute the score for all edges that are not part of the positive or negative training edges.
As we saw in section 3.3.1.1, two hyperparameters can be tuned with the Katz index:

- *max_path_length*: maximal length of the paths, i.e., the walks, to consider. The default is 5. For hyperparameter tuning, we consider the values 2, 3, 4 and 5.
- *damping_value*: value used to exponentially damp every addend of the sum. It should be in (0,1] and was referred to above as β. The default is 0.005. We use the values 0.0005, 0.005, 0.05 and 0.5 for hyperparameter tuning.

Table 45 summarizes all hyperparameters to be tuned with their respective values considered. In total, there are 16 hyperparameter combinations for which we fit the Katz index to our data with 10 different runs each.
For each hyperparameter combination, the results from the 10 runs are saved in the dataset

Hyperparameter	Values considered
max_path_length	2, 3, 4, 5
damping_value	0.0005, 0.005, 0.05, 0.5
Total combinations: 16	

Table 45: Katz index: hyperparameters

`ka_results.pkl`. There are two hyperparameter combinations that lead to the highest mean validation average precision of 0.9520. They produce the same results for all 10 runs and are given in Table 46.

Hyperparameter	Comb. 1: best value	Comb. 2: best value
max_path_length	5	0.0005
damping_value	5	0.005

Table 46: Katz index: best hyperparameter combinations

Table 47 shows the mean AP and mean AUC over the 10 runs in the validation and test set with the corresponding standard deviations for the two best hyperparameter combinations.

	AP	AUC
Validation	0.9520 ± 0.0064	0.9494 ± 0.0079
Test	0.9513 ± 0.0029	0.9485 ± 0.0037

Table 47: Katz index: mean APs and mean AUCs with standard deviations

For all 6 heuristics, very similar values are obtained for the mean validation AP and the mean test AP as well as for the mean validation AUC and the mean test AUC, which validates the respective performance on the test set.

We continue with the second part of our baselines, the latent-feature methods. In this context, we apply spectral clustering as a matrix factorization method as well as DeepWalk and node2vec as network embedding methods, where DeepWalk is a special case of node2vec. We do not apply content-based methods.

4.6.7 Spectral clustering

In the context of spectral clustering, we factorize the observed adjacency matrix of the HCP graph into the product of low-dimensional embeddings and their transpose.
Spectral clustering is implemented in the notebook `spectral_clustering_baseline.ipynb`, based on Hu (2018). As with the heuristics, we first perform the train-validation-test split and thus obtain the positive and negative training, validation, and test edges. We also obtain the adjacency matrix of the training graph, from which we create the training graph in networkx. With scikit-learn (Pedregosa et al. (2011)), we can learn spectral embeddings based on an adjacency matrix, which can be used for spectral clustering. The function needed for this is `spectral_embedding`, which has as main arguments the adjacency matrix and the dimension of the embeddings.
For spectral clustering as a whole, only the hyperparameter *n_components*, the dimension of the projection subspace, is to be tuned. Its default is 8. We consider the values 4, 8, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48, 52, 56, 60 and 64 for hyperparameter tuning.
So, there are 16 values for one hyperparameter for which we perform spectral clustering with 10 different runs each.
For each value of the hyperparameter, the results from the 10 runs are saved in the dataset `sc_results.pkl`. The value of 20 for the hyperparameter leads to the highest mean validation average precision of 0.7349.
Table 48 shows the mean AP and mean AUC over the 10 runs in the validation and test set with the corresponding standard deviations for best hyperparameter value of 20.

	AP	AUC
Validation	0.7349 ± 0.0267	0.7980 ± 0.0231
Test	0.7342 ± 0.0158	0.7966 ± 0.0122

Table 48: Spectral clustering: mean APs and mean AUCs with standard deviations

The mean validation AP and mean test AP as well as the mean validation AUC and mean test AUC are very similar, confirming the performance on the test set.

4.6.8 DeepWalk

DeepWalk is a method that learns low-dimensional embeddings for nodes using a skip-gram model based on random-walk-generated node sequences, whereby nodes that are close to each other in a random walk have similar representations. The pairwise node embeddings

are then aggregated as edge representations for link prediction.

Our implementation of DeepWalk draws from Hu (2018) and can be found in the notebook `deep_walk_baseline.ipynb`. As before, we first perform the train-validation-test split and thus obtain the positive and negative training, validation, and test edges. We also obtain the adjacency matrix of the training graph, from which we create the training graph in networkx.

Based on the functions in the file `node2vec_functions.py`, we now generate the random walks. With the help of the function `Word2Vec` from the gensim library (Řehůřek and Sojka (2010)) we can calculate the required embeddings, in this context also called word vectors. We can now generate edge embeddings by simply multiplying one-by-one the two corresponding node embeddings. In this way, we obtain corresponding edge embeddings for the positive and negative training, validation, and test edges. From the learned edge embeddings we now only have to predict whether a link exists between two nodes or not. For this we proceed as follows: We train a logistic regression classifier[44] as an edge classifier by regressing the training edges, which consist of both positive and negative training edges, on the training embeddings, which also consist of both positive and negative embeddings. We now use this trained classifier to obtain a prediction for all validation and test edges as to whether a link exists between two nodes or not.

Together with the validation and test edges we can calculate our metrics AP and AUC.

In DeepWalk, we tune the following hyperparameters:

- *window_size*: maximum distance between the current and predicted word within a sentence. This is a hyperparameter used in the optimization of the embeddings, i.e., the word vectors. For hyperparameter tuning, we consider the values 5 and 10.
- *num_walks*: number of walks starting from one node. We consider the values 10 and 20 for hyperparameter tuning.
- *walk_length*: length of walk starting from one node. The values 40 and 80 are used for hyperparameter tuning.
- *dimensions*: dimension of the embeddings, i.e., the word vectors. For hyperparameter tuning, we use the values 16, 32, 64 and 128.

There are other hyperparameters, but we do not tune them. These are:

[44] In contrast to node classification, no hyperparameter tuning takes place for the logistic regression classifier.

- *p*: return parameter. It controls the probability to go back to parts in the graph already visited in a random walk. Since we do not have any bias in DeepWalk, we set it to 1.
- *q*: in-out parameter: It controls the probability to go explore undiscovered parts in the graph. Since DeepWalk is not biased in which part of the graph it explores next, we set it to 1.
- *min_count*: all words with total frequency lower than this are ignored. We set this hyperparameter to 0.
- *sg*: determines the training algorithm. If set to 1, skip-gram is used. If set to 0, CBOW, i.e., continuous bag of words, is used. It is set to 1.
- *iter*: number of iterations, i.e., epochs, over the corpus. In newer versions of the library, it is called *epochs*. We set it to 1.

Table 49 summarizes all hyperparameters to be tuned with their respective values considered.

Hyperparameter	Values considered
window_size	5, 10
num_walks	10, 20
walk_length	40, 80
dimensions	16, 32, 64, 128
p	1 (fixed)
q	1 (fixed)
min_count	0 (fixed)
sg	1 (fixed)
iter	1 (fixed)
Total combinations: 32	

Table 49: DeepWalk: hyperparameters

In total, there are 32 hyperparameter combinations for which we fit DeepWalk to our data with 10 different runs each.
For each hyperparameter combination, the results from the 10 runs are saved in the dataset `dw_results.pkl`. The best hyperparameter combination leads to the highest mean validation average precision of 0.8151. It is given in Table 50.
Table 51 shows the mean AP and mean AUC over the 10 runs in the validation and test set with the corresponding standard deviations for the best hyperparameter combination.

Hyperparameter	Best value
window_size	5
num_walks	20
walk_length	80
dimensions	128
p	1 (fixed)
q	1 (fixed)
min_count	0 (fixed)
sg	1 (fixed)
iter	1 (fixed)

Table 50: DeepWalk: best hyperparameter combination

	AP	AUC
Validation	0.8151 ± 0.0196	0.8456 ± 0.0153
Test	0.8083 ± 0.0197	0.8417 ± 0.0141

Table 51: DeepWalk: mean APs and mean AUCs with standard deviations

We see that the values of the mean validation AP and mean test AP as well as the mean validation AUC and mean test AUC are very close, confirming the performance on the test set.

4.6.9 node2vec

Building on our previous work on DeepWalk, we now consider node2vec. It generalizes DeepWalk in that biased random walks, in this case second-order biased random walks, are now also permissible.

We base our implementation of node2vec in the notebook `node2vec_baseline.ipynb` on Hu (2018). As before, we first perform the train-validation-test split and thus obtain the positive and negative training, validation, and test edges. We also obtain the adjacency matrix of the training graph, from which we create the training graph in networkx.

Leveraging the functions in the file `node2vec_functions.py`, we now generate the second-order biased random walks. This bias in the random walks is induced by the two hyperparameters p and q. Here, p influences the probability to revisit already visited parts of the graph, while q influences the probability to visit not yet visited parts of the graph (Cohen (2018)). With the help of the function `Word2Vec` from the gensim library (Řehůřek and Sojka (2010)), we can calculate the required embeddings, in this context also called word vectors. We can now generate edge embeddings by simply multiplying one-by-one the two

corresponding node embeddings. In this way, we obtain corresponding edge embeddings for the positive and negative training, validation, and test edges. From the learned edge embeddings we now only have to predict whether a link exists between two nodes or not. For this we proceed as follows: we train a logistic regression classifier[45] as an edge classifier by regressing the training edges, which consist of both positive and negative training edges, on the training embeddings, which also consist of both positive and negative embeddings. We now use this trained classifier to obtain a prediction for all validation and test edges as to whether a link exists between two nodes or not.
Together with the validation and test edges we can calculate our metrics AP and AUC.
In node2vec, we tune the following hyperparameters:

- p: return parameter. It controls the probability to go back to parts in the graph already visited in a random walk. For hyperparameter tuning, we choose the values 1, 2, 3, 4 and 5.

- q: in-out parameter: It controls the probability to go explore undiscovered parts in the graph. We consider the values 1, 2, 3, 4 and 5 for hyperparameter tuning.

Hyperparameter tuning with node2vec is very computationally intensive. We decide only to tune p and q, the two hyperparameters that control the bias of the random walks. For the other four hyperparameters considered for tuning in DeepWalk, we decide to set them to the values that they take in the best hyperparameter combination for DeepWalk. Overall, the other hyperparameters that we do not tune in node2vec are:

- *window_size*: maximum distance between the current and predicted word within a sentence. This is a hyperparameter used in the optimization of the embeddings, i.e., the word vectors. We consider the value 5 from the best DeepWalk hyperparameter combination.

- *num_walks*: number of walks starting from one node. We take the value 20 from the best DeepWalk hyperparameter combination.

- *walk_length*: length of walk starting from one node. The value 40 from the best DeepWalk hyperparameter combination is used.

- *dimensions*: dimension of the embeddings, i.e., the word vectors. We use the value 128 from the best DeepWalk hyperparameter combination.

[45] Unlike in node classification, no hyperparameter tuning takes place for the logistic regression classifier.

- *min_count*: all words with total frequency lower than this are ignored. We set this hyperparameter to 0.
- *sg*: determines the training algorithm. If set to 1, skip-gram is used. If set to 0, CBOW, i.e., continuous bag of words, is used. We set it to 1.
- *iter*: number of iterations over the corpus. In newer versions of the library, it is called *epochs*. It is set to 1.

Table 52 summarizes all hyperparameters to be tuned with their respective values considered.

Hyperparameter	Values considered
p	1, 2, 3, 4, 5
q	1, 2, 3, 4, 5
window_size	5 (fixed)
num_walks	20 (fixed)
walk_length	40 (fixed)
dimensions	128 (fixed)
min_count	0 (fixed)
sg	1 (fixed)
iter	1 (fixed)
Total combinations: 25	

Table 52: node2vec: hyperparameters

In total, there are 25 hyperparameter combinations for which we fit node2vec to our data with 10 different runs each.
For each hyperparameter combination, the results from the 10 runs are saved in the dataset `nv_results.pkl`. The best hyperparameter combination leads to the highest mean validation average precision of 0.8894. It is given in Table 53.

Hyperparameter	Best value
p	1
q	5
window_size	5 (fixed)
num_walks	20 (fixed)
walk_length	40 (fixed)
dimensions	128 (fixed)
min_count	0 (fixed)
sg	1 (fixed)
iter	1 (fixed)

Table 53: node2vec: best hyperparameter combination

Table 54 shows the mean AP and mean AUC over the 10 runs in the validation and test set with the corresponding standard deviations for the best hyperparameter combination.

	AP	AUC
Validation	0.8894 ± 0.0173	0.9079 ± 0.0139
Test	0.8876 ± 0.0097	0.9071 ± 0.0049

Table 54: node2vec: mean APs and mean AUCs with standard deviations

The mean validation AP and mean test AP as well as the mean validation AUC and mean test AUC are very similar, validating the performance on the test set.

After applying our baselines, we turn our focus to GNNs, which can incorporate node features into link prediction in addition to the graph structure.

4.6.10 Graph autoencoder

The GAE learns embeddings for all nodes from its local neighborhood using a GCN and aggregates these node embeddings later to edge representations, which can then be used to predict links.

The notebook `gae_model_gcn_encoder.ipynb` contains our entire implementation of the GAE, which is based on the code provided by Fey and Lenssen (2019) on GitHub. With the PyTorch Geometric implementation of a GAE, the only arguments left to specify are the encoder and the decoder. For the encoder, we choose a GCN that has only one hidden layer in addition to the input layer and output layer. First and foremost, we aim to avoid overfitting. The input dimension is 68 according to the number of node features. The hidden dimension and the output dimension can both be tuned. Additionally, the dropout probability, which is used when applying dropout after the hidden layer, can also be tuned. By default, the inner product decoder is used as our decoder.

We use the cross entropy loss here, being already part of the PyTorch Geometric GAE implementation, and choose Adam as the optimizer. The learning rate and weight decay can also be tuned here.

In addition, we apply early stopping to reduce overfitting and to stop the training process if the validation loss does not decrease after a predefined number of epochs.

For each run, the losses and metrics are saved for the best epoch, i.e., the epoch with the lowest validation loss.

The main hyperparameters to tune are:

- *hidden_channels*: dimension of the embeddings in the hidden layer. The values considered for hyperparameter tuning are 16 and 32.
- *out_channels*: dimension of the embeddings in output layer. For hyperparameter tuning, the values 8 and 16 are used.
- *dropout_prob*: probability of a node to be dropped. We consider the values 0.1, 0.3 and 0.5 for hyperparameter tuning.
- *learning_rate*: learning rate of the Adam optimizer. Here, we use 0.001, 0.01 and 0.1 for hyperparameter tuning.
- *weight_decay*: L2 penalty added to the loss function to prevent overfitting. For hyperparameter tuning, we take the values 5e-1, 5e-2, 5e-3 and 5e-4 into account.
- *patience*: number of epochs after which to stop the training process if no decrease in the validation loss has been observed in the meantime. We take the values 3, 5 and 10 into account for hyperparameter tuning.

Table 55 summarizes all hyperparameters to be tuned with their respective values considered.

Hyperparameter	Values considered
hidden_channels	16, 32
output_channels	8, 16
dropout_prob	0.1, 0.3, 0.5
learning_rate	0.001, 0.01, 0.1
weight_decay	5e-1, 5e-2, 5e-3, 5e-4
patience	3, 5, 10
Total combinations: 432	

Table 55: Graph autoencoder: hyperparameters

There is a total of 432 hyperparameter combinations for which we fit the GAE to our data with 10 different runs each.

For each hyperparameter combination, the results from the 10 runs are saved in the dataset `gae_results.pkl`. It contains the best APs and AUCs from the 10 runs for the training, validation and test set for each hyperparameter combination and also the respective means and standard deviations over the 10 runs for the training, validation and test set.

There are three hyperparameter combinations that lead to the highest mean best validation average precision of 0.9303. They lead to exactly the same best training APs and AUCs, best validation APs and AUCs and mean best test APs and AUCs over the 10 runs, so that

the means and standard deviations of the best APs and AUCs in the training, validation and test set are also the same. Table 56 provides the three best hyperparameter combinations.

Hyperparameter	Comb. 1: best value	Comb. 2: best value	Comb. 3: best value
hidden_channels	16	16	16
out_channels	8	8	8
dropout_prob	0.5	0.5	0.5
learning_rate	0.001	0.001	0.001
weight_decay	0.05	0.05	0.05
patience	3	5	10

Table 56: Graph autoencoder: best hyperparameter combinations

Table 57 shows the means and standard deviations of the best APs and AUCs over the 10 runs in the training, validation and test set for the three best hyperparameter combinations.

	Best AP	Best AUC
Training	0.9364 ± 0.0020	0.9251 ± 0.0029
Validation	0.9303 ± 0.0069	0.9170 ± 0.0078
Test	0.9327 ± 0.0035	0.9193 ± 0.0053

Table 57: Graph autoencoder: mean best APs and AUCs with standard deviations

It is evident that the mean best AP and AUC in the training set are only slightly higher with values of 0.9364 and 0.9251, respectively, than the ones in the validation and test set. However, it is noticeable that the best APs and AUCs in the training set have by far the lowest standard deviations with values of 0.0020 and 0.0029. The mean best AP and AUC in the validation set are 0.9303 and 0.9170, respectively, while the mean best AP and AUC in the test set are 0.9327 and 0.9193, respectively. The two mean best APs and AUCs are close to each other and thus the performance on the validation set confirms that the performance on the test set is valid. Overall, the standard deviations of the best APs and AUCs in the validation and test set are a lot higher than in the training set, the respective values being 0.0069 and 0.0078 as well as 0.0035 and 0.0053.

All in all, we obtain a GAE that contains the following components

```
GAE(
    (encoder): GCNEncoder (
        (conv1): GCNConv(68, 16)
```

```
        (conv2): GCNConv(16, 8)
      )
      (decoder): InnerProductDecoder()
    ).
```

Table 58 shows how the total parameters of the GAE are distributed among its individual components.

Layer.Parameter	Parameter tensor shape	Number of parameters
encoder.conv1.bias	[16]	16
encoder.conv1.lin.weight	[16, 68]	1088
encoder.conv2.bias	[8]	8
encoder.conv2.lin.weight	[8, 16]	128
Total params: 1240		
Trainable params: 1240		
Non-trainable params: 0		

Table 58: GAE model parameters

Of the total 1240 parameters, the vast majority of $(68 + 1) \cdot 16 = 1104$ parameters belong to the first graph convolutional operator, the remaining $(16 + 1) \cdot 8 = 136$ parameters to the second graph convolutional operator.

To better understand the training process, we now take a look at how the losses, APs and AUCs develop over the epochs for the best hyperparameter combinations. The corresponding figures can be found in appendix E.

Figure 33 plots how the losses in the training, validation and test set develop over the epochs for the best hyperparameter combinations.
It can be seen that, other than with the node classification methods, the training process is never aborted by early stopping. This is particularly clear from the loss curves. They run almost like a straight downward line for the 10 runs, with no plateau being reached by the losses in learning. This may suggest that a higher number of epochs would be useful in this model for the loss to approach a plateau. The training losses are very variable compared to the validation and test losses for the 10 runs, which may be due to the high dropout probability of 0.5 for the best hyperparameter combinations. Overall, it can also be observed that the decrease in losses over the 100 epochs is rather moderate, generally from about 1.3 to

about 1.1. Training, validation and test losses are all about the same.

Figure 34 shows how the average precisions in the training, validation and test set develop over the epochs for the best hyperparameter combinations.
The APs are relatively high from the first epoch and always start at least with a value of 0.85. Despite large looking differences in the plots between the individual AP curves, the APs in the training, validation and test set are very close to each other for all 10 runs. This indicates that the GAE does not overfit.

Figure 35 visualizes how the areas under the curve in the training, validation and test set develop over the epochs for the best hyperparameter combinations.
The AUCs are relatively high from the first epoch and always start at least with a value of 0.89. Although there are large looking differences in the plots between the individual AUC curves, the AUCs in the training, validation and test set are very close to each other for all 10 runs. This supports our previous finding that the GAE does not overfit.

4.6.11 Variational graph autoencoder

The VGAE is the variational version of the GAE. It deploys two GCNs to learn the mean and the standard deviation of deterministic embeddings and uses the embedding means of two nodes to predict a link between them.
We implement the VGAE in the notebook `vgae_model_vgcn_encoder.ipynb` and build our implementation on the code provided by Fey and Lenssen (2019) on GitHub. Using the PyTorch Geometric implementation of a VGAE, we only still have to specify the encoder, decoder, mu and logstd. For the encoder, we choose a GCN that has only one hidden layer following the input layer. Again, we are keen to avoid overfitting. However, other than in any GNN considered before, there are two output layers: one output layer for mu and one output layer for logstd. The input dimension is 68 according to the number of node features. The hidden dimension and the output dimension, which is valid for both output layers, can be tuned. Additionally, the dropout probability, which is used when applying dropout after the hidden layer, can also be tuned. The inner product decoder is our decoder by default. For mu and logstd, we also use the defaults, whereby the last computations of mu and logstd are used.
As loss function, we use the sum of the cross entropy loss and the KL loss here. We choose Adam as the optimizer. The learning rate and weight decay can also be tuned here.

Additionally, we apply early stopping to reduce overfitting and to stop the training process if the validation loss does not decrease after a predefined number of epochs.
For each run, the losses and metrics are saved for the best epoch, i.e., the epoch with the lowest validation loss.
The main hyperparameters to tune are:

- *hidden_channels*: dimension of the embeddings in the hidden layer. We consider the values 16 and 32 for hyperparameter tuning.
- *out_channels*: dimension of the embeddings in the output layers. The values considered for hyperparameter tuning are 8 and 16.
- *dropout_prob*: probability of a node to be dropped. For hyperparameter tuning, we use the values 0.1, 0.3 and 0.5.
- *learning_rate*: learning rate of the Adam optimizer. Here, we use 0.001, 0.01 and 0.1 for hyperparameter tuning.
- *weight_decay*: L2 penalty added to the loss function to avoid overfitting. For hyperparameter tuning, we take the values 5e-1, 5e-2, 5e-3 and 5e-4 into account.
- *patience*: number of epochs after which to stop the training process if no decrease in the validation loss has been observed in the meantime. We take the values 3, 5 and 10 into account for hyperparameter tuning.

Table 59 provides a summary of all hyperparameters to be tuned with their respective values considered.

Hyperparameter	Values considered
hidden_channels	16, 32
output_channels	8, 16
dropout_prob	0.1, 0.3, 0.5
learning_rate	0.001, 0.01, 0.1
weight_decay	5e-1, 5e-2, 5e-3, 5e-4
patience	3, 5, 10
Total combinations: 432	

Table 59: Variational graph autoencoder: hyperparameters

In total, there are 432 hyperparameter combinations for which we fit the VGAE to our data with 10 different runs each.

For each hyperparameter combination, the results from the 10 runs are saved in the dataset `vgae_results.pkl`. It contains the best APs and the best AUCs from the 10 runs for the training, validation and test set for each hyperparameter combination and also the respective means and standard deviations over the 10 runs for the training, validation and test set. The best hyperparameter combination leads to the highest mean best validation average precision of 0.9299. It is given in Table 60.

Hyperparameter	Best value
hidden_channels	32
out_channels	16
dropout_prob	0.5
learning_rate	0.001
weight_decay	0.005
patience	10

Table 60: Variational graph autoencoder: best hyperparameter combination

Table 61 shows the means and standard deviations of the best APs and mean best AUCs over the 10 runs in the training, validation and test set for the best hyperparameter combination.

	Best AP	Best AUC
Training	0.9354 ± 0.0025	0.9228 ± 0.0049
Validation	0.9299 ± 0.0080	0.9167 ± 0.0097
Test	0.9315 ± 0.0034	0.9167 ± 0.0049

Table 61: Variational graph autoencoder: mean best APs and AUCs with standard deviations

As becomes obvious, the mean best AP and AUC in the training set are only slightly higher with values of 0.9354 and 0.9228, respectively, than the ones in the validation and test set. However, it is noticeable that the best APs have by far the lowest standard deviation with a value of 0.0025. For the best AUCs, the standard deviation is as high as for the best AUCs in the test set. The mean best AP and AUC in the validation set are 0.9299 and 0.9167, respectively, while the mean best AP and AUC in the test set are 0.9315 and 0.9167, respectively. The two mean best APs and AUCs are close to each other and thus the performance on the validation set validates the performance on the test set. Overall, the standard deviations of the best APs and AUCs in the validation and the best APs in the test set are a lot higher than in the training set, the respective values being 0.0080 and 0.0097 as well as 0.0034.

Overall, the best APs and AUCs obtained with the VGAE are only minimally lower than the ones obtained with the GAE. Apart from very small deviations, both models show the same performance.

Overall, we have a VGAE with the following components

```
VGAE(
    (encoder): VariationalGCNEncoder(
        (conv1): GCNConv(68, 32)
        (conv_mu): GCNConv(32, 16)
        (conv_logstd): GCNConv(32, 16)
    )
    (decoder): InnerProductDecoder()
).
```

Table 62 shows how the total parameters of the VGAE are distributed among its individual components.

Layer.Parameter	Parameter tensor shape	Number of parameters
encoder.conv1.bias	[32]	32
encoder.conv1.lin.weight	[32, 68]	2176
encoder.conv_mu.bias	[16]	16
encoder.conv_mu.lin.weight	[16, 32]	512
encoder.conv_logstd.bias	[16]	16
encoder.conv_logstd.lin.weight	[16, 32]	512
Total params: 3264		
Trainable params: 3264		
Non-trainable params: 0		

Table 62: VGAE model parameters

Of the total 3264 parameters, the majority of $(68 + 1) \cdot 32 = 2208$ parameters belong to the first graph convolutional operator, the remaining $2 \cdot (32 + 1) \cdot 16 = 1056$ parameters to the two second graph convolutional operators.

In a bid to better understand the training process, we now consider the development of the losses, APs and AUCs over the epochs for the best hyperparameter combination, where the corresponding figures can be found in appendix E.

Figure 36 shows how the losses in the training, validation and test set develop over the epochs for the best hyperparameter combination.
We can see that the training loss decreases constantly for all runs and does not reach a discernible plateau for most runs. As with the GAE, the training loss is relatively variable for all runs, which is again due to the high dropout probability of 0.5 for the best hyperparameter combination. The validation and test loss are much smaller than the training loss for all runs from the first epoch onwards. They either decrease only slightly over the epochs, as for runs 3, 7, 8 and 9, or are almost constant, as for runs 1, 2, 4, 5, 6 and 10, so that the training process stops after 20 to 40 epochs.

Figure 37 visualizes how the average precisions in the training, validation and test set develop over the epochs for the best hyperparameter combination.
The APs are relatively high from the first epoch and always start at least with a value of 0.91. Despite large looking differences in the plots between the individual AP curves, the APs in the training, validation and test set are very close to each other for all 10 runs, which shows us that the VGAE does not overfit.

Figure 38 plots how the areas under the curve in the training, validation and test set develop over the epochs for the best hyperparameter combination.
The AUCs are relatively high from the first epoch and always start at least with a value of 0.89. In spite of large looking differences in the plots between the individual AUC curves, the AUCs in the training, validation and test set are very close to each other for all 10 runs. This again confirms our statement that the VGAE does not overfit.

4.6.12 SEAL

Finally, we turn to our most complex model for link prediction: SEAL. Being a subgraph-based method, an enclosing subgraph is extracted for each target link to be predicted before applying a graph-level GNN with pooling that predicts the existence of a link based on the enclosing subgraph.
Our implementation of SEAL can be found in the notebook `seal_model.ipynb` and relies on the code provided by Fey and Lenssen (2019) on GitHub. Different from the GAE and the VGAE, there is no implementation for SEAL in PyTorch Geometric. For this reason, implementing SEAL is more complex. Since SEAL leverages the subgraphs around

two nodes between which a link is to be predicted, we create a `SEALDataset` from the `LinkPredictionDataset` that gives us the k-hop enclosing subgraphs for all positive and negative training, validation, and test edges, assigning a label to each node in the respective subgraph based on DRNL and processing these resulting features into one-hot features. It should be noted that we do not concatenate these one-hot features with the other node features, because our goal is to learn links from structure while dropping node features. Based on this, we implement the DGCNN, i.e., the graph-level GNN, into which the enclosing subgraphs and the one-hot features are fed to learn a graph classification function. As described in Zhang et al. (2018), we fit the DGCNN as follows: A subgraph is passed through a number of graph convolutional layers so that node information can propagate between neighbors. Then, the global pooling operator is applied, where node features are sorted in descending order based on their last feature channel. The number of nodes for each subgraph is then standardized, where we truncate the sorted sequence of nodes if a previously calculated value is smaller than the number of nodes and pad otherwise. After that, a 1D convolution is applied, followed by a 1D max pooling. Then, a 1D convolution is applied again and the result is passed to an MLP. Here, the number of convolutional layers as well as the hidden dimension of the convolutional layers can be tuned.
As loss function, we use the binary cross entropy with logits loss. We choose Adam as the optimizer. The learning rate and weight decay can also be tuned here. We also apply early stopping to reduce overfitting and to stop the training process if the validation loss does not decrease after a predefined number of epochs.
For each run, the losses and metrics are saved for the best epoch, i.e., the epoch with the lowest validation loss. Note in particular that we only run SEAL for one single hyperparameter combination because the model is extremely computationally intensive.
In general, the main hyperparameters to tune are:

- *num_hops*: specifies the maximum number of hops that a node is allowed to be away in order to be still part of the enclosing subgraph. We consider the value 2 for hyperparameter tuning.

- *hidden_channels*: dimension of each convolutional layer in the DGCNN. The value considered for hyperparameter tuning is 32.

- *num_layers*: number of convolutional layers in the DGCNN. We take the value 3 for hyperparameter tuning.

- *learning_rate*: learning rate of the Adam optimizer. Here, we use 0.0001 for hyperpa-

rameter tuning.

- *weight_decay*: L2 penalty added to the loss function to prevent overfitting. For hyperparameter tuning, we take the value 0 into account.
- *patience*: number of epochs after which to stop the training process if no decrease in the validation loss has been observed in the meantime. We choose the value 3 for hyperparameter tuning.

Table 63 gives the one hyperparameter combination for which SEAL is fit.

Hyperparameter	Value considered
num_hops	2
hidden_channels	32
num_layers	3
learning_rate	0.0001
weight_decay	0
patience	3
Total combinations: 1	

Table 63: SEAL: hyperparameters

So, in total, there is only 1 hyperparameter combination for which we fit SEAL to our data with 10 different runs each.
The results for this hyperparameter combination are saved in the dataset `seal_results.pkl`. It contains the best APs and the best AUCs from the 10 runs for the training, validation and test set for the one hyperparameter combination and also the respective means and standard deviations over the 10 runs for the training, validation and test set.
The corresponding mean best validation average precision is given by 0.9763.

	Best AP	Best AUC
Training	0.9821 ± 0.0038	0.9816 ± 0.0038
Validation	0.9763 ± 0.0057	0.9761 ± 0.0053
Test	0.9764 ± 0.0034	0.9771 ± 0.0033

Table 64: SEAL: mean best APs and AUCs with standard deviations

We observe that the mean best AP and AUC in the training set are only minimally higher with values of 0.9821 and 0.9816, respectively, than the ones in the validation and test set. The mean best AP and AUC in the validation set are 0.9763 and 0.9761, respectively, while the mean best AP and AUC in the test set are 0.9764 and 0.9771, respectively. The two

mean best APs and AUCs are close to each other, validating the performance on the test set. We also notice that the best APs and AUCs in the test set have the lowest standard deviations with values of 0.0034 and 0.0033, respectively. The standard deviations of the best APs and AUCs in the training set are not much higher with values of 0.0038 and 0.0038, respectively, whereas the best APs and AUCs in the validation set with values of 0.0057 and 0.0053, respectively, are noticeably higher.

All in all, we obtain a DGCNN, the GNN used by SEAL, with the following components

```
DGCNN(
    (convs): ModuleList(
        (0): GCNConv(16, 32)
        (1): GCNConv(32, 32)
        (2): GCNConv(32, 32)
        (3): GCNConv(32, 1)
    )
    (conv1): Conv1d(1, 16, kernel_size=(97,), stride=(97,))
    (maxpool1d): MaxPool1d(kernel_size=2, stride=2, padding=0, dilation=1,
        ceil_mode=False)
    (conv2): Conv1d(16, 32, kernel_size=(5,), stride=(1,))
    (mlp): MLP(3360, 128, 1)
).
```

Table 65 shows how the total parameters of the DGCNN are distributed among its individual components.
Of the total 437186 parameters, the vast majority of $(3360+1)\cdot 128+(128+1)\cdot 1 = 430337$ parameters belong to the MLP at the end of the DGCNN, the remaining 6849 parameters to the four convolutional layers and the two 1D convolutions before.

To better understand the training process, we now look at the development of the losses, APs and AUCs over the epochs. The corresponding figures can be found in appendix E.

Figure 39 visualizes how the losses in the training, validation and test set develop over the epochs for the one hyperparameter combination.

Layer.Parameter	Parameter tensor shape	Number of parameters
convs.0.bias	[32]	32
convs.0.lin.weight	[32, 16]	512
convs.1.bias	[32]	32
convs.1.lin.weight	[32, 32]	1024
convs.2.bias	[32]	32
convs.2.lin.weight	[32, 32]	1024
convs.3.bias	[1]	1
convs.3.lin.weight	[1, 32]	32
conv1.weight	[16, 1, 97]	1552
conv1.bias	[16]	16
conv2.weight	[32, 16, 5]	2560
conv2.bias	[32]	32
mlp.lins.0.weight	[128, 3360]	430080
mlp.lins.0.bias	[128]	128
mlp.lins.1.weight	[1, 128]	128
mlp.lins.1.bias	[1]	1
Total params: 437186 Trainable params: 437186 Non-trainable params: 0		

Table 65: DGCNN model parameters

We observe that for all runs the training process always ends before epoch 30 and is thus aborted relatively quickly. The training loss always starts at a value of about 0.4 and then steadily decreases without fluctuations to a value of about 0.2. From epoch 10 onwards, it is the smallest of all losses for all runs. The validation and test loss also decrease steadily, but show smaller fluctuations.

Figure 40 plots how the average precisions in the training, validation and test set develop over the epochs for the one hyperparameter combination.
The APs are relatively high from the first epoch and always start at least with a value of 0.93. Even though there are larger looking differences in the plots between the individual AP curves, the APs in the training, validation and test set are very close to each other for all 10 runs. Therefore, we conclude that SEAL does not overfit.

Figure 41 shows how the areas under the curve in the training, validation and test set develop over the epochs for the one hyperparameter combination.
The AUCs are relatively high from the first epoch and always start at least with a value of 0.92. Despite larger looking differences in the plots between the individual AUC curves, the

AUCs in the training, validation and test set are very close to each other for all 10 runs. This confirms that SEAL does not overfit.

4.6.13 Summary and comparison

In a bid to finalize our link prediction task, we give a brief summary and comparison of the performance of all methods - baselines and GNNs - on the test set.

Table 66 shows the means and standard deviations of the APs and AUCs in the test set realized for the heuristics in the 10 runs. These are visualized by the boxplots in Figure 12.

Method	AP	AUC
Common neighbors	0.9479 ± 0.0031	0.9494 ± 0.0036
Jaccard	0.9545 ± 0.0027	0.9545 ± 0.0027
Preferential attachment	0.9306 ± 0.0037	0.9159 ± 0.0045
Adamic-Adar	0.9573 ± 0.0028	0.9567 ± 0.0034
Resource allocation	0.9703 ± 0.0025	0.9692 ± 0.0031
Katz	0.9513 ± 0.0029	0.9485 ± 0.0037

Table 66: Link prediction: results of heuristics

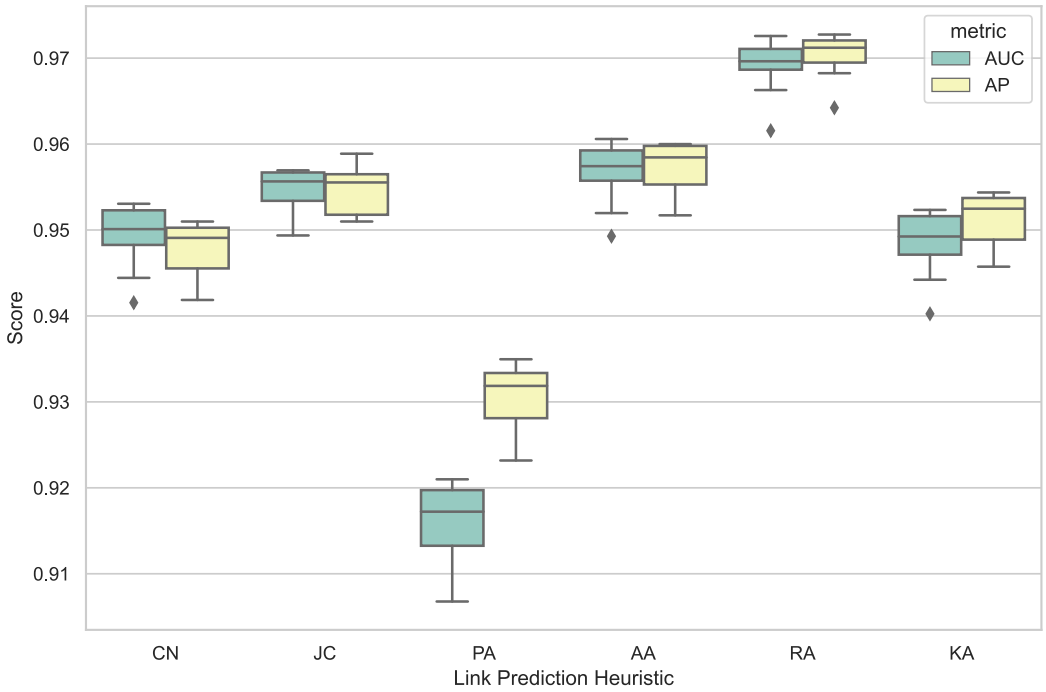


Figure 12: Link prediction: boxplots of heuristics results

By a noticeable margin, resource allocation gives the best performance on the test set in terms of both metrics because its mean test AP and mean test AUC are the highest. In addition, its standard deviation of the test APs over the 10 runs is also the lowest of all six heuristics, whereas its standard deviation of the test AUCs over the 10 runs is the second lowest after the Jaccard score, so that overall resource allocation also offers a very reliable performance. Nevertheless, we note that the other heuristics - apart from preferential attachment, which is a bit further behind - show a similarly good performance.

Table 67 gives us the means and standard deviations of the APs and AUCs in the test set realized for the latent-feature methods in the 10 runs. The boxplots in Figure 13 plot these.

Method	AP	AUC
Spectral clustering	0.7342 ± 0.0158	0.7966 ± 0.0122
DeepWalk	0.8083 ± 0.0197	0.8417 ± 0.0141
node2vec	0.8876 ± 0.0097	0.9071 ± 0.0049

Table 67: Link prediction: results of latent-feature methods

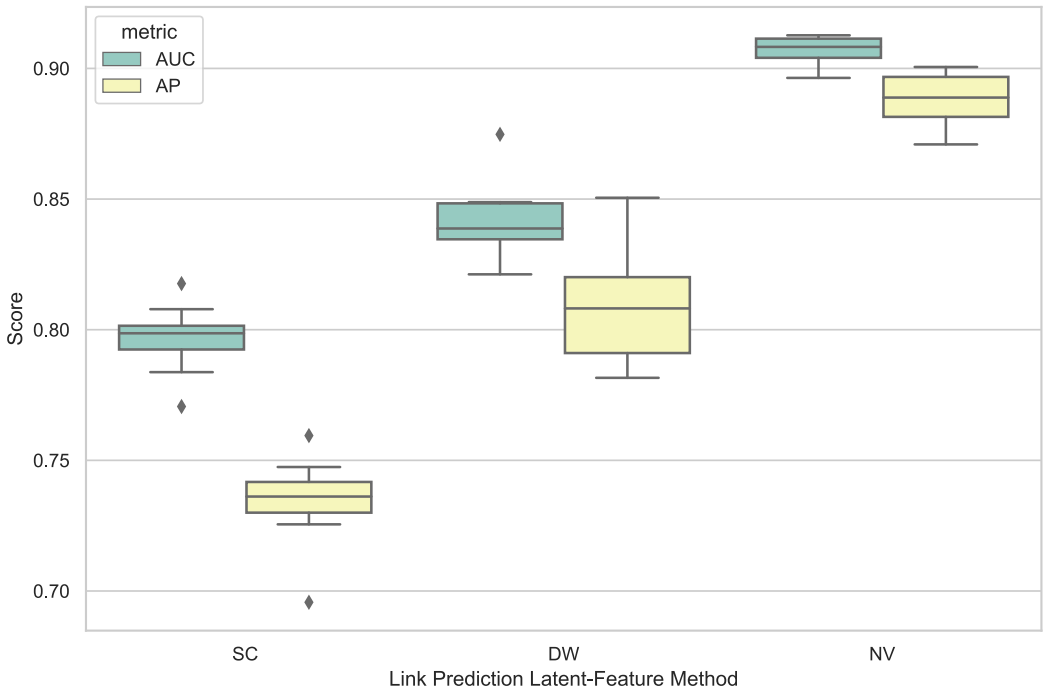


Figure 13: Link prediction: boxplots of latent-feature methods results

We see that among the latent-feature methods, node2vec performs best on the test set. Not only does it give the highest values for the mean test AP and mean test AUC, but it also has the most reliable results because the test APs and test AUCs over the 10 runs have the lowest standard deviations. DeepWalk scores noticeably worse than node2vec both in terms of mean test AP and mean test AUC and in turn spectral clustering gives a significantly poorer performance than DeepWalk with regard to both mean test AP and mean test AUC.

Table 68 contains the means and standard deviations of the best test APs and AUCs for the GNNs in the 10 runs, which are visualized by the boxplots in Figure 14.

Method	Best AP	Best AUC
GAE	0.9327 ± 0.0035	0.9193 ± 0.0053
VGAE	0.9315 ± 0.0034	0.9167 ± 0.0049
SEAL	0.9764 ± 0.0034	0.9771 ± 0.0033

Table 68: Link prediction: results of GNNs

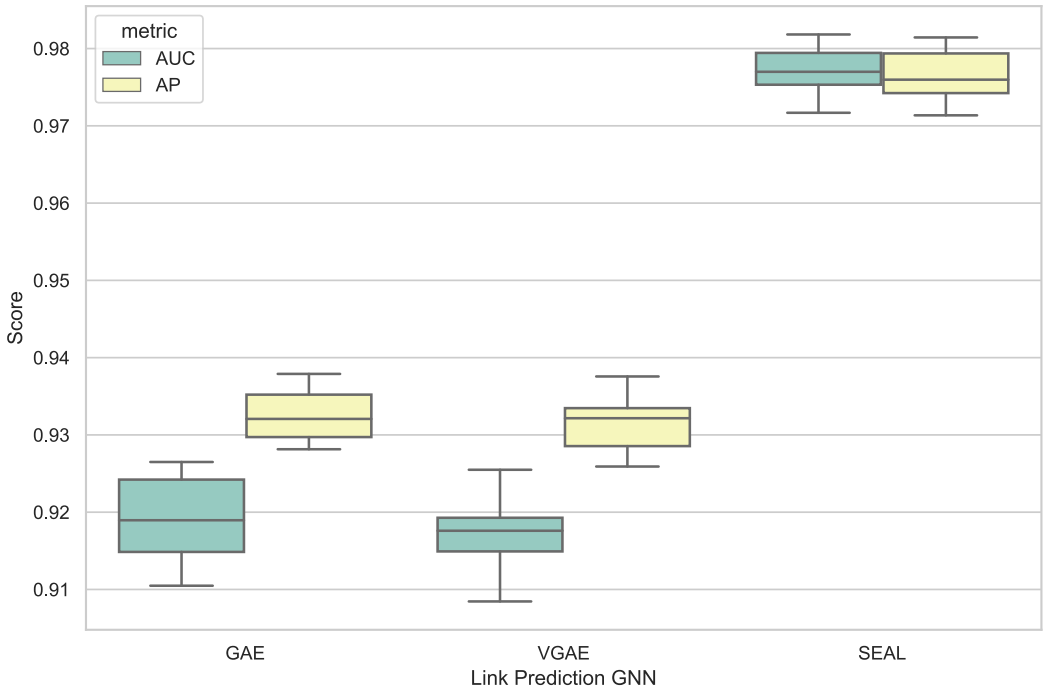


Figure 14: Link prediction: boxplots of GNN results

In general, all GNNs demonstrate a good performance. However, SEAL stands out as the

best model both in terms of the mean best test AP and the mean best test AUC. The performance of the GAE and VGAE is virtually the same, with minimal deviations in the mean best test AP and the mean best test AUC in favor of the GAE. In addition, the standard deviations of the best test APs and best test AUCs are relatively low, so that all three GNNs produce reliable results.

In conclusion, considering our link prediction task as a whole, SEAL is the best performing model due to giving the highest values for both metrics. The heuristics also show a convincing performance, while the latent-feature methods perform worst. It should be noted that resource allocation as a heuristic is only slightly behind the SEAL in spite of being so much simpler. Overall, SEAL as a GNN proves the superior performance that GNNs can achieve compared to simpler methods in general and classical machine learning methods in particular.

4.7 Fourth task: time series forecasting

Within the scope of time series forecasting it is our objective to simultaneously forecast the sales of two cancer drugs contained in 57 Novartis sales time series. Thus, our task belongs to the field of multivariate time series forecasting. These 57 time series are divided into the 35 time series in the BC regions with sales of Kisqali and the 22 time series in the melanoma regions with sales of the drug combination Tafinlar-Mekinist. For all time series, the sales are given for 24 months from 09-2019 to 08-2021.
Research has already shown that the inclusion of a graph in forecasting that learns the relationships between the individual time series is eventually reflected in more accurate forecasts. Our motivation is to find out whether this potential also manifests itself in the forecasting of our time series.
Since we have a total of 57 time series for 24 months for one feature, the sales, we obtain a [1, 24, 57] tensor that forms the basis of this task. It is saved in the file `sales.h5`. Depending on the different methods we use in the course of this task, this time series data is processed differently according to the requirements of the respective methods. The further processing steps are always described below in the explanations of the individual methods.
Overall, our goal is to always create forecasts exclusively for the next month. Thus, our forecasting horizon is 1. We do the train-validation-test split in such a way that we have the sales of the first 15 months, i.e., from 09-2019 to 11-2020, in the training set, the sales of the following 3 months, i.e., from 12-2020 to 02-2021, in the validation set and the sales

of the last 6 months, i.e., from 03-2021 to 08-2021, in the test set. Using the validation set, we validate the performance on the test set. We report the mean absolute error (MAE), mean absolute percentage error (MAPE) and root mean square error (RMSE) as performance metrics. Note that the time series are very short, which makes it very difficult to learn structures in them. In contrast, the time series in SHANG ET AL. (2021), for example, where the superiority of GTS was shown on the basis of two datasets, are much longer: For the two datasets, the time series had a length of greater than 30,000 and 50,000, respectively. In addition, from our time series plots provided in Figures 42 to 51 in appendix F, we can spot some irregularities[46]: Some time series fluctuate a lot, while others almost consistently take the value 0. In order to be able to fit all models to the given time series without any problems, some values of the time series with index 20, i.e., the Kisqali sales in the BC region Norrbotten-Sunderbyn which contain a lot of sales of 0, are changed from 0 to 1. We do this with the time series values with indices 0 (09-2019), 3 (12-2019), 8 (05-2020), 15 (12-2020) and 17 (02-2021). Of course, all models are fit to this slightly modified data.

We adopt the following procedure. First, we fit three baselines to the data to obtain forecasts for the last 9 months, i.e., for the validation and test set. These baselines include static forecasting, historical average forecasting and vector autoregression and are supposed to enable us to evaluate the performance of GTS on the test set at the end of this task. More precisely, we verify if GTS which incorporates a graph structure of dependencies between time series into forecasting succeeds in outperforming the baselines on the test set which cannot consider such type of dependencies between time series. Then, we fit GTS to the time series and get forecasts for the last 9 months. After that, we provide a short summary of our results and establish which method is most performant.

In contrast to node classification and link prediction, we have here a non-random train-validation-test split, so that performing one run for each model is sufficient. Replicability of our results is ensured by seeding the training process. All metric values below are rounded to four decimal places. The best hyperparameter combination is the one that gives the lowest validation MAE respectively the lowest best validation MAE.[47]

For each method, we document precisely how it is applied. For GTS, we additionally explain which hyperparameters are tuned, which hyperparameter combination performs best and what the overall best forecasting performance is. The development of losses and metrics is also documented.

[46] The actual sales referred to here are shown in blue in the corresponding figures.

[47] For the baselines, there is only one value for the MAE, MAPE and RMSE. For GTS as a GNN, however, we receive the different MAEs, MAPEs and RMSEs over all epochs, so that we report the best achieved values for the MAE, MAPE and RMSE.

We now get started with the baselines, which comprise static forecasting, historical average forecasting and vector autoregression.

4.7.1 Static forecasting

Static forecasting uses a lagged value of the variable to be forecast as the forecast value. This is done for each time series individually. In our application, we want to forecast only for the next month, i.e., the horizon equals 1. Therefore, we consider only lags of 1, i.e., monthly lags. This means, for example, that in order to forecast the sales for 06-2021 in Blekinge, we use the sales from 05-2021 in Blekinge as the sales forecast.
The notebook `static_baseline.ipynb` contains our entire implementation of static forecasting, which relies on the code provided by Shang et al. (2021) on GitHub. There are no hyperparameters to tune. We can directly apply static forecasting on the 57 sales time series in `sales.h5` because no further transformation of the time series is needed. The forecasts for the validation set are saved in `y_predict_val_st.pkl`, while the forecasts for the test set are saved in `y_predict_test_st.pkl`.
Table 69 shows the three performance metrics MAE, MAPE and RMSE for the validation and test set.

	MAE	MAPE	RMSE
Validation	77.53	0.6545	142.18
Test	92.17	0.5435	172.27

Table 69: Static forecasting: MAEs, MAPEs and RMSEs

All metrics show very high values. The poor performance is particularly evident in the MAPE, because the forecasts deviate from the true sales by an average of 65.45% in the validation set and 54.35% in the test set.

4.7.2 Historical average forecasting

The method of historical average forecasting makes use of periodicity to come up with forecast values. For example, given a periodicity of 3 months in the time series, if we want to generate a forecast for 06-2021 in Blekinge, we would use the average of the sales from 03-2021, 12-2020, 09-2020, 06-2020, 03-2020, 12-2019 and 09-2019 as the sales forecast. This is done for each time series separately. In our application, we want to forecast only for the next month, i.e., the horizon equals 1. Since the time series, which are plotted in blue in

Figures 42 to 51 in appendix F, do not show any periodicity, we decide to simply take the average of all past sales as the forecast. This implies a periodicity of 1. That means, for example, that in order to forecast the sales for 06-2021 in Blekinge, we would use the average of the sales from 09-2019 to 05-2021 as the sales forecast.
In the notebook `historical_average_baseline.ipynb`, we provide our entire implementation of historical average forecasting, which is based on the code made available by Shang et al. (2021) on GitHub. We do not have to tune any hyperparameters. Like static forecasting, historical average forecasting does not require any transformation of the time series to be applied. We save the forecasts for the validation set in `y_predict_val_ha.pkl` and the forecasts for the test set in `y_predict_test_ha.pkl`.
The three performance metrics MAE, MAPE and RMSE for the validation and test set are given in Table 70.

	MAE	MAPE	RMSE
Validation	85.76	0.6636	152.25
Test	93.04	0.5564	163.90

Table 70: Historical average forecasting: MAEs, MAPEs and RMSEs

The metrics show even higher values than with static forecasting, the only exception being a lower RMSE in the test set. Here, the poor performance is again evident considering the MAPE. The forecasts deviate from the true sales by an average of 66.36% in the validation set and 55.64% in the test set.

4.7.3 Vector autoregression

Vector autoregression (VAR) is a statistical model that allows for multivariate time series, i.e., multiple time series observed at the same point in time. Therefore, it can be regarded as the more-dimensional analog of autoregressive models. It can account for the relationship between multiple variables that change over time. The VAR model can be modeled as a system of equations, where each variable, here a sales time series, gets an equation modeling its evolution over time. This equation includes the variable's lagged values, the lagged values of the other variables in the model, and an error term. Here, the other variables in the model are the other time series (Dissanayake (2020)). In our application, we want to forecast only for the next month, i.e., the horizon equals 1. We decide to only consider lags of 1, i.e., monthly lags, and therefore a VAR model of order 1. For the implementation of vector autoregression, we resort to the implementation available in the statsmodels library

(Seabold and Perktold (2010)), which fits the model and enables forecasting. As opposed to static forecasting and historical average forecasting, we now need to transform the time series appropriately by standardizing them so that they have a mean of 0 and a standard deviation of 1. Based on these standardized time series, the VAR model is first trained using the time series data in the training set. Then, the forecasts for the validation and test set are generated with the trained VAR model. Eventually, these forecasts are transformed back using the inverse transformation, resulting in the final forecasts.
Our implementation of vector autoregression rests on the code released by Shang et al. (2021) on GitHub and is given in the notebook `vector_autoregression_baseline.ipynb`. Setting the order of the VAR model to 1, there are no hyperparameters left to tune. We save the forecasts for the validation set in `y_predict_val_va.pkl` and the forecasts for the test set in `y_predict_test_va.pkl`.
Table 71 contains the three performance metrics MAE, MAPE and RMSE for the validation and test set.

	MAE	MAPE	RMSE
Validation	117.40	1.3957	219.29
Test	121.57	0.7644	226.59

Table 71: Vector autoregression: MAEs, MAPEs and RMSEs

The metrics show even higher values than with static forecasting and historical average forecasting. Here, the MAPE illustrates the poor performance still more clearly than before. The forecasts deviate from the true sales by an average of 139.57% in the validation set and 76.44% in the test set, which makes them useless.

Knowing how our three baselines perform, we now look into the implementation of GTS. We are keen to find out if taking into account the interdependencies between the time series in the form of a graph enhances forecasting performance with respect to the three metrics.

4.7.4 Graph for time series

GTS proposed by Shang et al. (2021) is an advanced GNN for multivariate time series forecasting which learns a graph with the relationships between the time series in the training process in order to exploit it for more accurate forecasts. If no graph structure between the time series is known in advance, then it learns a graph during the training process. In case a graph structure between the time series is known upfront, GTS can use this as a

priori knowledge to improve the graph it learns even more.

The complete implementation of GTS takes place in the notebook `gts_model.ipynb`. However, the generation of the training, validation and test data is outsourced to the notebook `train_val_test_generation.ipynb`, where the time series from `sales.h5` are put into the form in which they are later fed into GTS. The resulting training, validation and test data is saved in `train.npz`, `val.npz` and `test.npz`, respectively. In overall terms, we make intensive use of the code that Shang et al. (2021) place at our disposal on GitHub. We modify this code and tailor it to our specific time series forecasting task. PyTorch Geometric does not provide an implementation of GTS. Therefore, the implementation is carried out with PyTorch and turns out to be extremely extensive. For this reason, we limit ourselves here to the central parts. First, we program the DCGRUCell and the Seq2Seq architecture, so that we can define the encoder model and the decoder model based on them. These correspond to the lower part of Figure 6 in section 3.4.2 illustrating the GTS architecture. With the help of the encoder and the decoder model, we then implement the GTS model, which corresponds to the entire GTS architecture, i.e., both the upper and lower part of in Figure 6. The GTS model is structured as follows: For the feature extractor, we have a 1D convolutional layer, followed by a 1D batch normalization layer, and then again a 1D convolutional layer. After that, we stack a fully-connected layer followed by a 1D batch normalization layer onto this. Then, we apply two fully connected layers for the link predictor. This is followed by the encoder model and finally the decoder model. In order for the training process to work properly, we integrate the GTS model into the GTS supervisor, in which everything flows together. In addition to containing the GTS model, the GTS supervisor is responsible for things such as loading and standardizing the time series, feeding them into the GTS model, creating the kNN neighborhood graph serving as a prior and learning the graph as well as transforming the forecasts of the GTS model back to obtain the final forecasts.

We deploy the MAE between the true sales values and the forecasts. As the kNN neighborhood graph serves as a prior, the MAE is extended by the BCE of θ and the prior A^{a}, using a regularization magnitude of $\lambda = 1$. Furthermore, we choose Adam as the optimizer. We also apply early stopping with a patience of 100 to reduce overfitting and to stop the training process if the validation loss does not decrease after this number of epochs.

For each hyperparameter combination, the losses and metrics are stored for the best epoch, i.e., the epoch with the lowest validation loss.

The behaviour of GTS is determined by a variety of parameters. Some of these parameters are also hyperparameters, which can either be tuned or be assigned a fixed value. We always

highlight when a parameter is a hyperparameter. If it is tuned, we provide the respective values considered for hyperparameter tuning and if it is assigned a fixed value, we provide this fixed value. In the next steps, the parameters of the GTS model are explained. They are divided into the three groups "Data", "Model" and "Train", depending on which part of the learning task they pertain to.

Data

- *batch_size*: training batch size, which we set to the number of forecasts we can make for the training set. Due to our horizon of 1, this is 14.
- *val_batch_size*: validation batch size. Choosing the number of forecasts we can make for the validation set, we set it to 3.
- *test_batch_size*: test batch size. It is set to 3, corresponding to the number of forecasts we can make for the test set.

Model

- *cl_decay_steps*: used for computing the sampling threshold. Although being a hyperparameter, we do not tune it and set it to 2000.
- *filter_type*: filter that is not used in our implementation but which can be used in future extensions of our model. For now, we set it to 'dual_random_walk'.
- *horizon*: number of months for which to forecast. We only want to forecast for the next month and thus set it to 1.
- *input_dim*: number of features to use for forecasting. Since we only use sales as features, it is set to 1.
- *max_diffusion_step*: diffusion degree for the DCGRUCell. This is a hyperparameter, which we set to 2.
- *num_nodes*: number of time series. Therefore, it takes a value of 57.
- *num_rnn_layers*: number of layers in the DCGRUCell used both for the encoder and the decoder. This is a hyperparameter, which we set to 1.
- *output_dim*: number of features to forecast. Given that we only have the sales as features, it is set to 1.

- *rnn_units*: number of units, i.e., neurons, in each of the layers in the DCGRUCell. This is a hyperparameter and we consider the values 16, 32 and 64 for hyperparameter tuning.
- *seq_len*: number of months to use for forecasting. Since we only use the current month to forecast for the next month, we set it to 1.
- *use_curriculum_learning*: whether to use curriculum learning, where the difficulty gradually increases. This is a hyperparameter, which we set to True.
- *dim_fc*: input dimension of the fully-connected layer at the end of the feature extractor. In order for dimensions to match, we need to set it to 176.

Train

- *base_lr*: learning rate of the optimizer. Here, we use 0.001, 0.01 and 0.1 for hyperparameter tuning.
- *epoch*: index of the first epoch. We set it to 0.
- *epochs*: number of epochs to use for training. This hyperparameter is set to 200.
- *epsilon*: term added to the denominator to improve numerical stability. We set this hyperparameter to 0.001. Its default value is 1e-8.
- *lr_decay_ratio*: multiplicative factor of learning rate decay. Hyperparameter tuning uses the values 0.05, 0.1, which is the default, and 0.2.
- *max_grad_norm*: max norm of the gradients when clipping the gradient norm of the model parameters. We set this hyperparameter to 5.
- *optimizer*: the optimizer to use. We use Adam as optimizer.
- *patience*: number of epochs after which training is discontinued if no reduction in validation loss has been achieved in the meantime. We decide to set this hyperparameter to 100.
- *steps*: indices of epochs at which to decay the learning rate. We use the epochs with the indices 20, 30 and 40 for this hyperparameter.
- *test_every_n_epochs*: specifies at how many epochs apart testing is performed. Since we intend to perform testing every epoch, we set it to 1.

- *knn_k*: specifies the number of neighbors for nodes in the kNN neighbors graph serving as prior. This hyperparamater is tuned with the values 3, 5 and 10.
- *epoch_use_regularization*: specifies up to which epoch regularization is performed. Given that we perform regularization during the whole training process, we set this hyperparameter to 200, the number of epochs.

Table 72 summarizes all parameters with their respective values considered.

Parameter	Values considered
Data	
batch_size	14
test_batch_size	6
val_batch_size	3
Model	
cl_decay_steps	2000
filter_type	'dual_random_walk'
horizon	1
input_dim	1
max_diffusion_step	2
num_nodes	57
num_rnn_layers	1
output_dim	1
rnn_units	**16, 32, 64**
seq_len	1
use_curriculum_learning	True
dim_fc	176
Train	
base_lr	**0.001, 0.01, 0.1**
epoch	0
epochs	200
epsilon	0.001
lr_decay_ratio	**0.05, 0.1, 0.2**
max_grad_norm	5
optimizer	'adam'
patience	100
steps	[20, 30, 40]
test_every_n_epochs	1
knn_k	**3, 5, 10**
epoch_use_regularization	200
Total combinations: 81	

Parameters whose corresponding values are printed in bold are hyperparameters for which hyperparameter tuning is performed with the corresponding values printed in bold.

Table 72: Graph for time series: parameters

In total, there are 81 hyperparameter combinations for which we fit GTS to our time series. For each hyperparameter combination, the results are saved in the dataset `gts_results.pkl`. It contains the best MAEs, the best MAPEs and the best RMSEs for the validation and test set for each hyperparameter combination.
There are three hyperparameter combinations that lead to the lowest best validation MAE of 73.27. They also lead to exactly the same best validation MAEs, MAPEs and RMSEs and best test MAEs, MAPEs and RMSEs and are given in Table 73.

Hyperparameter	Comb. 1: best value	Comb. 2: best value	Comb. 3: best value
rnn_units	64	64	64
base_lr	0.01	0.01	0.01
lr_decay_ratio	0.05	0.1	0.2
knn_k	3	3	3

Table 73: Graph for time series: best hyperparameter combinations

Table 74 shows the best MAE, best MAPE and best RMSE for the validation and test set achieved with the best hyperparameter combinations.

	Best MAE	Best MAPE	Best RMSE
Validation	73.27	0.4998	108.14
Test	83.65	0.4405	148.53

Table 74: Graph for time series: best MAEs, MAPEs and RMSEs

All three metrics in the test set clearly show that GTS delivers significantly better forecasts than the three baselines. The best test MAE of 83.65 of GTS is significantly lower than the lowest test MAE of the baselines of 92.17, which is achieved by static forecasting. Moreover, the best test MAPE of 0.4405 indicates that GTS performs significantly better than static forecasting, which attains the lowest test MAPE of the baselines with a value of 0.5435. The difference in performance is just as clear considering the RMSE. The lowest test RMSE of the baselines is achieved by historical average forecasting and takes a value of 163.90. GTS is able to undercut this value with a best test RMSE of 148.53.
Similar results are obtained for the performance on the validation set.

Overall, our GTS model is composed of the following layers

```
GTSModel(
```

```
    (conv1): Conv1d(1, 8, kernel_size=(3,), stride=(1,))
    (bn1): BatchNorm1d(8, eps=1e-05, momentum=0.1, affine=True,
        track_running_stats=True)
    (conv2): Conv1d(8, 16, kernel_size=(3,), stride=(1,))
    (bn2): BatchNorm1d(16, eps=1e-05, momentum=0.1, affine=True,
        track_running_stats=True)
    (fc): Linear(in_features=176, out_features=100, bias=True)
    (bn3): BatchNorm1d(100, eps=1e-05, momentum=0.1, affine=True,
        track_running_stats=True)
    (fc_out): Linear(in_features=200, out_features=100, bias=True)
    (fc_cat): Linear(in_features=100, out_features=2, bias=True)
    (encoder_model): EncoderModel(
        (dcgru_layers): ModuleList(
            (0): DCGRUCell()
        )
    )
    (decoder_model): DecoderModel(
        (projection_layer): Linear(in_features=64, out_features=1, bias=True)
        (dcgru_layers): ModuleList(
            (0): DCGRUCell()
        )
    )
).
```

Table 75 shows how the total parameters of GTS are distributed among its individual layers. Of the total 114011 parameters, a total of $17600 + 100 + 20000 + 100 + 200 + 2 = 38002$ parameters belong to the fully-connected layers of the feature extractor and link predictor. Nearly all other parameters belong to the encoder and decoder.

To better understand the training process, we now take a closer look at the development of the losses, MAEs, MAPEs and RMSEs over the epochs. The corresponding figures can be found in Appendix F.

Figure 52 reveals how the loss in the training, validation and test set develops over the

(1)	(2)	(3)
conv1.weight	[8, 1, 3]	24
conv1.bias	[8]	8
bn1.weight	[8]	8
bn1.bias	[8]	8
conv2.weight	[16, 8, 3]	384
conv2.bias	[16]	16
bn2.weight	[16]	16
bn2.bias	[16]	16
fc.weight	[100, 176]	17600
fc.bias	[100]	100
bn3.weight	[100]	100
bn3.bias	[100]	100
fc_out.weight	[100, 200]	20000
fc_out.bias	[100]	100
fc_cat.weight	[2, 100]	200
fc_cat.bias	[2]	2
encoder_model.dcgru_layers.0.gconv_weight_(195, 128)	[195, 128]	24960
encoder_model.dcgru_layers.0.gconv_biases_128	[128]	128
encoder_model.dcgru_layers.0.gconv_weight_(195, 64)	[195, 64]	12480
encoder_model.dcgru_layers.0.gconv_biases_64	[64]	64
decoder_model.projection_layer.weight	[1, 64]	64
decoder_model.projection_layer.bias	[1]	1
decoder_model.dcgru_layers.0.gconv_weight_(195, 128)	[195, 128]	24960
decoder_model.dcgru_layers.0.gconv_biases_128	[128]	128
decoder_model.dcgru_layers.0.gconv_weight_(195, 64)	[195, 64]	12480
decoder_model.dcgru_layers.0.gconv_biases_64	[64]	64
Total params: 114011		
Trainable params: 114011		
Non-trainable params: 0		

The columns contain the following information: (1): Layer.Parameter, (2): Parameter tensor shape, (3): Number of parameters.

Table 75: GTS model parameters

epochs for the best hyperparameter combinations.

We can see that training, validation and test loss reach a plateau below 100 from around epoch 50 onwards. Initially, the training loss is the highest by a small margin, followed by the test loss. The smallest is the validation loss. These differences become smaller and smaller with each additional epoch. However, we observe some fluctuations in the losses.

Figure 53 plots how the MAE in the validation and test set develops over the epochs for the best hyperparameter combinations.

A comparison of the validation and test MAEs with the validation and test losses in Figure

52 suggests that they are equal. In fact, however, they differ by a constant smaller than 1. This is the additional loss induced by the regularization, the second part of the loss in GTS, used to ensure that the learned graph is a variant of the a priori graph. As we see, the regularization hardly has any impact here. The pattern that we observe is the same as for the validation and test losses: from about epoch 50 onwards, the validation and test MAEs reach a plateau below 100. At the beginning, the validation MAE is marginally smaller than the test MAE. In later epochs, however, the validation and test MAE are about the same size. Of course, we observe the same fluctuations in the MAEs as in the losses.

Figure 54 shows how the MAPE in the validation and test set develops over the epochs for the best hyperparameter combinations.
Also for the validation and test MAPE it can be seen that they approach a plateau from about epoch 50 onwards. The initial values of the MAPEs are very high: the validation MAPE is initially slightly below 4.5, while the test MAPE is initially slightly above 3 and thus clearly below. These larger differences of the validation and test MAPE become significantly smaller with increasing epochs, but are still noticeable after reaching the plateau.

Figure 55 visualizes how the RMSE validation and test set develops over the epochs for the best hyperparameter combinations.
The validation RMSE is slightly above 400 at the beginning, while the test RMSE is around 500 there. Here, we observe that a plateau is reached from around epoch 10 onwards. The validation RMSE is then mostly below 150, while the test RMSE is a little above 150. We observe strong fluctuations in the validation RMSE.

Finally, let us consider again the two graphs that have been used in GTS: the a priori graph, here a kNN neighborhood graph, and the graph learned by the GTS model. Both are meant to represent the connections among the time series. In particular, the graph learned by the GTS model is leveraged to generate better forecasts.
Figure 56 displays the a priori graph, which is a kNN neighborhood graph, while Figure 57 depicts the graph learned by the GTS model. Table 82 and Table 83 provide a summary for the a priori graph and the learned graph, respectively.
We immediately notice that the a priori graph has significantly fewer links than the learned graph and consists of two connected components. Thus, the a priori graph conveys that there are two groups of time series which do not influence each other. Only time series

within a connected component can influence each other.
The learned graph, on the other hand, has over 6 times as many edges as the a priori graph and thus looks much denser. There is only one connected component. Therefore, the learned graph uses significantly more connections between the time series in the forecasts than the a priori graph would.
How can there be such differences between the two graphs if the a priori graph is supposed to serve as a prior for the learned graph? The reason is surely the weakness of the regularization. We have chosen a regularization magnitude of 1 for λ, so that the MAE and regularization loss are just added for the overall loss. We have also seen that the regularization loss is below 1 and thus represents a only a very small constant in comparison to the MAE. Hence, the regularization loss is not very important for the overall loss and cannot provide a real regularization, which would make the learned graph similar to the a priori graph. Consequently, it would be very enlightening to find out how systematic variation of the regularization magnitude affects the nature of the learned graph and the GTS performance as captured by the three metrics.

4.7.5 Summary and comparison

We finish our time series forecasting task by shortly summarizing and comparing the performance of the three baselines and GTS on the test set. At this point, it should be noted once again that in this task all metrics take relatively large values, which is due to the nature of the underlying time series data.

Table 76 gives us the MAEs, MAPEs, and RMSEs of the three baselines on the test set.

Method	MAE	MAPE	RMSE
Static forecasting	92.17	0.5435	172.27
Historical average forecasting	93.04	0.5564	163.90
Vector autoregression	121.57	0.7644	226.59

Table 76: Time series forecasting: results of baselines

Of the three baselines, static forecasting performs best on the test set by a small margin in terms of MAE and MAPE. Historical average forecasting is with a larger distance the strongest baseline on the test set with respect to the RMSE. Vector autoregression shows by far the worst performance on the test set for all three metrics.

Table 77 reports the best MAE, best MAPE, and best RMSE of GTS on the test set.

Method	Best MAE	Best MAPE	Best RMSE
Graph for time series	83.65	0.4405	148.53

Table 77: Time series forecasting: results of GTS

It is recognizable that GTS produces significantly lower values than the three baselines for each metric. Thus, GTS is the best of our time series forecasting models. This provides conclusive evidence that learning a graph reflecting the interdependencies between the time series - possibly based on an a priori graph - and incorporating it into time series forecasting possesses great potential for more accurate forecasts.

5 Conclusion

In the present master's thesis, the focus was on a research project that comprised three GNN-related tasks. These tasks were completed on the basis of company data. The main goal was to compare the performance of different GNNs on these three learning tasks with the performance of a rich set of baselines using appropriate metrics and to establish whether the GNNs can outperform the baselines. With this exploratory orientation, we wanted to determine whether the potential that GNNs have already demonstrated on many benchmark datasets also transfers to our data. As preliminary work for this, we started by laying the formal and content-related foundations for working with graphs. Afterwards, the general functionality of GNNs was presented. On this basis, the GNNs for the three GNN-related tasks were set forth.

Important findings

In general, the results are satisfactory and have confirmed the finding described in the literature that better predictions can be generated across different learning tasks by using GNNs than by using classical statistical methods. Furthermore, the values of the performance metrics calculated on the respective test sets could generally be validated by the values of these performance metrics on the validation sets. Hyperparameter tuning always played an important role: Depending on the hyperparameters chosen, the performance of the same model could vary drastically.
In the context of node classification, we suspected that the inclusion of the HCP graph would lead to better predictions for the segment of individual HCPs. We observed that in node classification all baselines showed good performance. The support vector classifier was the model with the best performance. Thus, good predictions could already be achieved without including the graph structure. As we have seen from the feature importances of the random forest, there were a few variables that already had a high explanatory power for the variable "segment". Contrary to our assumption, none of the three GNNs could outperform the support vector classifier. Only the GNN with Chebyshev convolution could demonstrate a performance close to that of the support vector classifier. The GCN and the GAT showed a very poor performance, comparable to that of random guessing. Obviously, the inclusion of the graph in the GCN and GAT led to a worse performance compared to the baselines. Why this was not the case with the GNN with Chebyshev convolution may be due to the fact that the Chebyshev filter size K in the best hyperparameter combination was 1 and

thus a kind of regularization was created, which prevented overfitting.
For link prediction, our goal was to find out whether the joint consideration of the graph structure and the node features could lead to an improved prediction power with respect to links. It turned out that the heuristics, the first part of the baselines, already showed a very good performance. This could be due to the fact that their underlying assumptions are fulfilled in our HCP graph. For example, the common neighbors heuristic assumes that the more common neighbors two nodes have, the more likely it is that there is a link between these two nodes. It is very likely that this assumption is satisfied for any two nodes in our HCP graph. For the latent-feature methods, the second part of the baselines, good results were still obtained, but they were no longer as good as those of the heuristics. As we learned in chapter 3.3.1.2, latent-feature methods need an extremely large dimension to represent simple heuristics. The embedding dimension is 128 for DeepWalk and node2vec, but only 20 for spectral clustering, which may explain why the performance of spectral clustering was significantly behind that of DeepWalk and node2vec. For the GNNs, we found that GAE and VGAE performed very well, roughly comparable to the performance of heuristics. The SEAL model not only turned out to be the best performing GNN, but also showed the best performance overall among all link prediction methods. It should be noted in particular that we only fit SEAL for one hyperparameter combination and thus an even better performance could possibly be achieved with a larger-scale hyperparameter tuning.
In time series forecasting, we wanted to ascertain whether a graph structure between the individual time series learned in the context of a GNN would lead to better forecasts compared to some classical time series forecasting methods as baselines. Among the baselines, static forecasting performed best for all three metrics. For our 57 time series with considerable fluctuations and many sales of 0, we found that this month's sales values were best predicted with last month's sales values. However, the values of the metrics as a whole were enormously high: the MAPE of 54.35% in the testing set for static forecasting indicated that the values predicted by static forecasting in the testing set deviated on average by 54.35% from the true values in the testing set. The GTS model managed to outperform the baselines in all three metrics. The values of the metrics in the GTS model were quite a bit lower than in static forecasting, but they were still quite high, as the MAPE of 44.05% showed.

Limitations of the findings

However, taking a holistic view of our research project, it should be noted that there were

deficiencies in the project data. In particular, the 57 time series that were the subject of the time series forecasting should be mentioned. With only 24 observations, they were extremely short and allowed the GTS model in particular only to a limited extent to learn structures in the time series and to use them for better forecasts. With longer time series data, a significant improvement of the forecasts and thus also of metrics such as the MAPE would be expected. In addition, there were massive differences between the individual time series: In some time series, sales fluctuated very strongly throughout, while in other time series they were almost consistently at 0. Especially in the latter case, where time series were almost constant, time series forecasting was made much more difficult.
Furthermore, some simplifications were made during the project. These include, in particular, the use of a homogeneous HCP graph, although this is actually a heterogeneous HCP graph. Using a heterogeneous HCP graph could be beneficial because prior learning tasks such as node classification could be performed more accurately. For example, we could learn whether explicitly the link type "joint paper" helps to predict an HCP's segment.
In particular, it must be taken into account that the models fit as part of the project were sometimes very complex and that often even one run of the model took a considerable amount of time. The runtimes accumulated especially against the background of hyperparameter tuning. Although we carried out wide-ranging hyperparameter tuning for our models overall, even more extensive hyperparameter tuning would be possible with greater available computational capacity, which would allow even better results to be achieved. Finally, it should be noted that the performance of the GNNs realized on our data is also only valid for our data and therefore does not allow any statement on how well the implemented GNNs would perform on other data.

So how powerful are GNNs in the final analysis?

Although GNNs were inferior to baselines in terms of node classification performance, we could still see the potential of GNNs by their convincing performance in link prediction and time series forecasting.
In particular, where a meaningful graph structure is available for the task at hand and where expressive features are present, GNNs coupled with extensive hyperparameter tuning can produce results that outperform those of heuristics and other classical machine learning methods. This finding is consistent with the insight from the relevant GNN literature.

6 Outlook

As an application of GNNs on three learning tasks leveraging enterprise data, our research project is based on the current state of research and can accordingly itself be placed within the current state of science. In addition, we were repeatedly confronted with obstacles in the course of the project, from which some future research directions can be derived.

Contribution of our research to knowledge in the field

Our findings highlight the importance of GNNs in learning tasks with an underlying graph structure. This master's thesis contributes to the current body of knowledge by showing not only how to fit and tune GNNs but also how to assess their performance and that of appropriate baselines in the context of three popular learning tasks. It also details the steps involved in the research project from obtaining the raw data to evaluating the results.
The research is significant because it demonstrates how methods whose potential has already been proven on benchmark datasets can be applied on real-world enterprise data. Thanks to the comprehensive documentation, the master's thesis becomes accessible to the layman and also provides a good introduction to GNNs in general. In addition, similarly oriented research projects can rely on our work in the future.

Future research directions

With respect to our research project, some extensions of our work could be addressed as future research directions. The most important extension is a more comprehensive data set. Especially in time series forecasting, it would be worthwhile to look at the performance of the implemented models on much longer time series, from which structures more useful for forecasting can be learned. For node classification in particular, we would also need more meaningful variables at HCP level to better predict the segment of an HCP. Finally, in both node classification and link prediction, we could consider transformations of variables to increase the predictive power of the features for the respective outcome variable.
In constructing our HCP graph, we extracted three different link types. Even though, strictly speaking, the HCP graph is a heterogeneous graph, we treated it as a homogeneous graph. The reason for this was that the GNNs developed so far are tailored to homogeneous graphs. Therefore, as a future research direction, a focus should be placed on GNNs for heterogeneous graphs, as Wu et al. (2021) also point out in their work. Such GNNs would also

be a tremendous asset to other research projects.

Although many useful implementations of common GNNs are already available in libraries such as PyTorch Geometric, it would be a great enrichment to extend this set of existing implementations, e.g., with implementations of GNNs such as SEAL or GTS. This would further simplify the implementation of GNNs, which in turn would make GNNs more accessible and more widely used.

The limited computational capacity has turned out to be a real bottleneck in the context of the research project, since hyperparameter tuning could only take place to a limited extent. Both in our research project and in other research projects, a higher computational capacity would ultimately lead to finding even better hyperparameter combinations and thus further improve the performance of GNNs. Wider availability of e.g. GPUs is thus of crucial importance.

Finally, another research direction could result from the application of GNNs on further tasks using again real-world data. These include, for example, the community detection (e.g. Yuan et al. (2022)) and graph classification (e.g. Wang et al. (2021)) tasks mentioned in section 2.2. It would be a great advance for research to ascertain which performance the respective GNNs show in comparison to suitable baselines outside benchmark datasets when applied on real-world data.

Appendix A

This glossary defines some selected terms that are not explicitly explained in the main part of the thesis but are nevertheless important for understanding.

Centrality: Centrality measures are measures of the relative importance of nodes in a graph. There are several measures of centrality, each defining importance differently. The most common four of them are degree centrality, eigenvector centrality, closeness centrality and betweenness centrality (Disney (2020)).

1. **Degree centrality:** With degree centrality, an importance score is assigned based on the number of links held by each node. It tells how many direct, "one-hop" connections each node has to other nodes in the network. It can be used to find very connected individuals, popular individuals, individuals who are likely to hold most information or individuals who can quickly connect with the wider network. Note that degree centrality is the simplest measure of node connectivity.

2. **Eigenvector centrality:** As with degree centrality, eigenvector centrality measures a node's importance based on the number of links it has to other nodes in the network. However, eigenvector centrality goes beyond degree centrality in that it also takes into account how well connected a node is, and how many links their connections have, and so on through the network. Since eigenvector centrality calculates the extended connections of a node, it can identify nodes with influence over the entire network, not just over those directly connected to it.

3. **Closeness centrality:** With closeness centrality, each node is scored based on their "closeness" to all other nodes in the network. This centrality measures the shortest paths between all nodes and then assigns each node a score based on its sum of shortest paths. It can be used to find individuals who are in the best position to influence the whole network most quickly.

4. **Betweenness centrality:** With betweenness centrality, the number of times a node lies on the shortest path between other nodes is measured. It shows which nodes are "bridges" between nodes in a network. It does this by identifying all the shortest paths and then counting how many times each node falls on one. Therefore, it can be used to find individuals who influence the flow around a system.

Clustering coefficient: Formally, the clustering coefficient is the number of closed triplets over the total number of triplets in a graph. This measure gives an indication of the clustering in the whole graph.

Connected component: A connected component is a connected subgraph that is not part of any larger connected subgraph.

Degree assortativity: Degree assortativity measures how well nodes with similar node degrees are connected to each other.

Density: In a graph of n nodes, the density is the ratio of the number of edges of the graph to the number of edges in a complete graph on n nodes.

Isolated: A node with degree zero is isolated.

Self-loop: A self-loop is an edge that connects a node to itself.

Subgraph: A subgraph is a graph whose nodes and edges are included in the nodes and edges of another graph.

Appendix B

In our research project, data on Swedish cities, hospitals and regions plays a crucial role. We want to ensure an understanding of the relationships between this data, which requires the provision of some additional information. In the light of this, we supply the following addenda:

- a table showing which Swedish city hosts which hospitals,
- a table on how the Swedish counties as officially defined by Swedish authorities relate to the Novartis-defined BC regions,
- a table on how the Swedish counties as officially defined by Swedish authorities relate to the Novartis-defined melanoma regions,
- a map of Sweden showing Sweden divided into the Swedish counties as officially defined by Swedish authorities.

To recall, Swedish melanoma regions bear the suffix "ONCO", whereas Swedish BC regions do not.

Hospitals in Swedish cities

Table 78 shows which hospitals with at least one HCP working there are located in which Swedish city. Note that university hospitals are highlighted in italics and that Skånes Universitetssjukhus Lund and Skånes Universitetssjukhus Malmö form Skånes Universitetssjukhus.

City	Hospitals
Borås	Södra Älvsborgs Sjukhus Borås
Eskilstuna	Mälarsjukhuset
Falun	Falu Lasarett
Gävle	Gävle Sjukhus
Göteborg	*Sahlgrenska Universitetssjukhuset*
Halmstad	Hallands Sjukhus Halmstad
Helsingborg	Helsingsborgs Lasarett
Jönköping	Länssjukhuset Ryhov
Kalmar	Länssjukhuset i Kalmar
Karlskrona	Blekingesjukhuset Karlskrona
Karlstad	Centralsjukhuset Karlstad
Lidköping	Skaraborgs Sjukhus Lidköping
Linköping	*Universitetssjukhuset i Linköping*
Lund	Lunds Universitet
	Skånes Universitetssjukhus Lund
Malmö	*Skånes Universitetssjukhus Malmö*
Skellefteå	Skellefteå Lasarett
Skövde	Skaraborgs Sjukhus Skövde
Solna	*Karolinska Universitetssjukhuset*
Stockholm	Drottningmottagningen
	Capio S:t Görans Sjukhus
	Södersjukhuset
	Sophiahemmet
Sundsvall	Länssjukhuset Sundsvall-Härnösand
Uddevalla	Uddevalla Sjukhus
Umeå	*Norrlands Universitetssjukhus*
Uppsala	*Akademiska Sjukhuset*
Varberg	Hallands Sjukhus Varberg
Visby	Visby Lasarett
Västerås	Västmanlands Sjukhus Västerås
Växjö	Centrallasarettet Växjö
Örebro	*Universitetssjukhuset Örebro*
Östersund	Östersunds Sjukhus

Table 78: Swedish cities and hospitals located there

Relationship between counties and BC regions

Table 79 shows which BC regions are part of which counties. The counties are defined much larger, so that often several BC regions form one county. Note that the BC region Uppsala is part of both the county Gävleborg and the county Uppsala. Thus, it is not always possible to unambiguously determine the associated county from the BC region.

County	BC regions
Blekinge	Blekinge
Dalarna	Dalarna
Gotland	Stockholm-Gotland
Gävleborg	Uppsala
	Gävleborg-Gävle
Halland	Halland-Halmstad
	Halland-Varberg-Falkenberg
	Västra Götaland-Göteborg
Jämtland	Jämtland
Jönköping	Jönköping-Jönköping
	Jönköping-Nässjö-Eksjö
	Jönköping-Värnamo
Kalmar	Kalmar
Kronoberg	Kronoberg-Ljungby
	Kronoberg-Växjö
Norrbotten	Norrbotten-Sunderbyn
Skåne	Skåne-Kristianstad
	Skåne-Helsingborg-Landskrona
	Skåne-Lund
Stockholm	Stockholm
Södermanland	Sörmland-Eskilstuna
Uppsala	Uppsala
Värmland	Värmland-Karlstad
Västerbotten	Västerbotten-Umeå
	Västerbotten-Skellefteå
Västernorrland	Västernorrland-Sundsvall
	Västernorrland-Örnsköldsvik
Västmanland	Västmanland-Västerås
Västra Götaland	Västra Götaland-Uddevalla
	Västra Götaland-Borås
	Västra Götaland-Lidköping
	Västra Götaland-Skövde
	Västra Götaland-Göteborg
	Västra Götaland-Alingsås
Örebro	Örebro-Örebro
Östergötland	Östergötland-Linköping
	Östergötland-Norrköping

Table 79: Mapping of BC regions to counties

Relationship between counties and melanoma regions

Table 80 shows which melanoma regions are part of which counties. The counties are defined larger, so that sometimes two melanoma regions form one county. Note that the melanoma region Västra Götaland-Göteborg ONCO is part of both the county Halland and the county Västra Götaland. Thus, it is not always possible to unambiguously determine the associated county from the melanoma region.

County	Melanoma regions
Blekinge	Blekinge ONCO
Dalarna	Dalarna ONCO
Gotland	Stockholm-Gotland ONCO
Gävleborg	Gävleborg-Gävle ONCO
Halland	Halland-Halmstad ONCO
	Västra Götaland-Göteborg ONCO
Jämtland	Jämtland ONCO
Jönköping	Jönköping ONCO
Kalmar	Kalmar ONCO
Kronoberg	Kronoberg-Växjö ONCO
Norrbotten	Norrbotten-Sunderbyn ONCO
Skåne	Skåne ONCO
Stockholm	Stockholm ONCO
Södermanland	Sörmland-Eskilstuna ONCO
Uppsala	Uppsala ONCO
Värmland	Värmland-Karlstad ONCO
Västerbotten	Västerbotten-Umeå ONCO
Västernorrland	Västernorrland-Sundsvall ONCO
Västmanland	Västmanland-Västerås ONCO
Västra Götaland	Västra Götaland-Göteborg ONCO
	Västra Götaland-SÄS ONCO
Örebro	Örebro-Örebro ONCO
Östergötland	Östergötland-Linköping ONCO

Table 80: Mapping of melanoma regions to counties

Map of Sweden

Figure 15 is a map of Sweden, showing where exactly in Sweden the counties are located. With the help of Table 79 and Table 80 from above we can see just as well where the BC regions and melanoma regions are located. In addition, all cities are shown where there is a hospital with at least one HCP working there. The code used to create the map is provided in `map_sweden.R` and is based on various code examples covered by Moreno and Basille (2018) as well as Herman (2020), which we have developed further significantly to satisfy our needs.

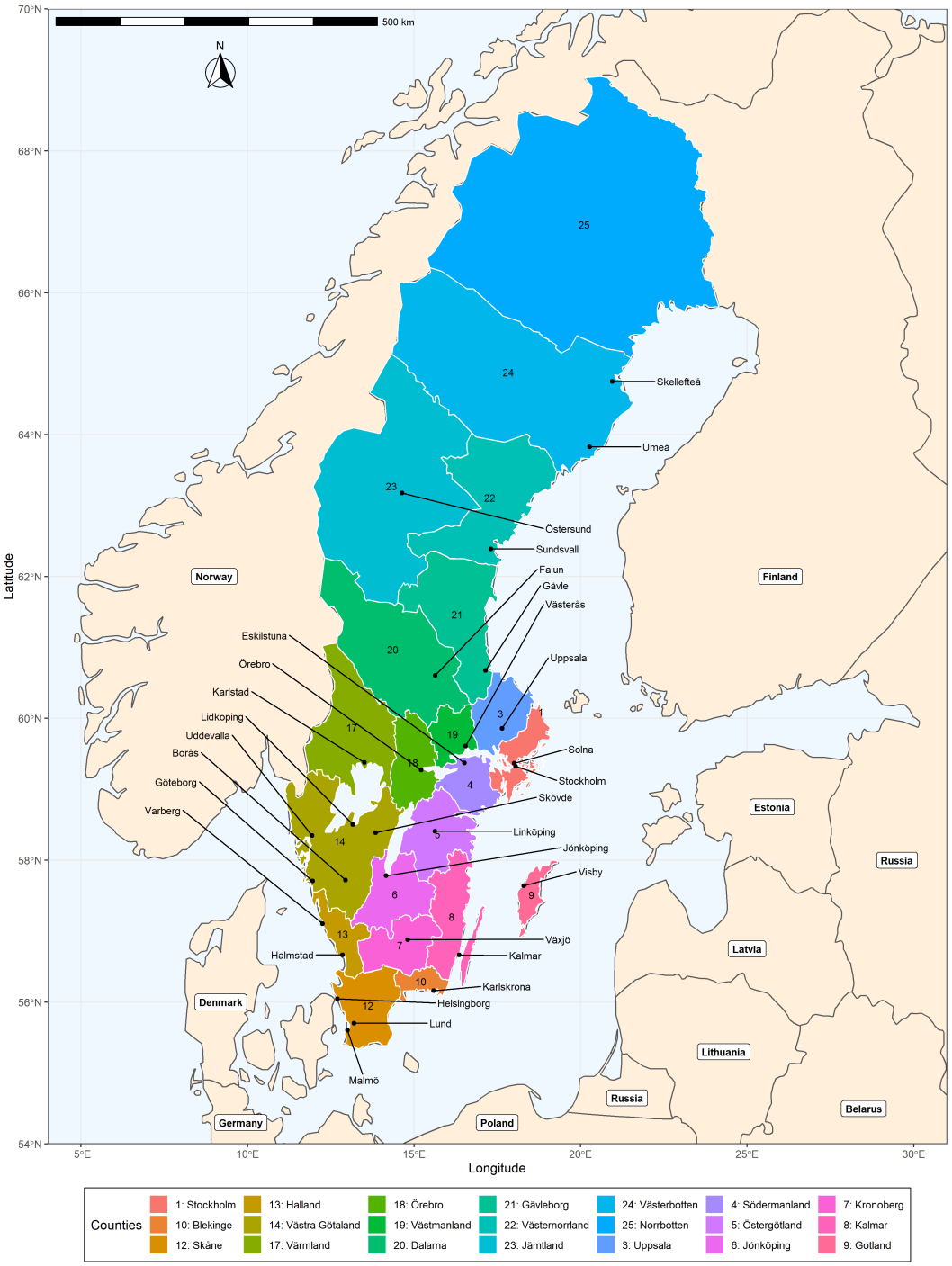


Figure 15: Map of Sweden

Appendix C

For both node classification and link prediction we leverage the node features. In this context, we have to transform the metric node features and therefore need to decide whether to use standardization or normalization as transformation. For this purpose, Figure 16 shows the distribution of the metric node features.

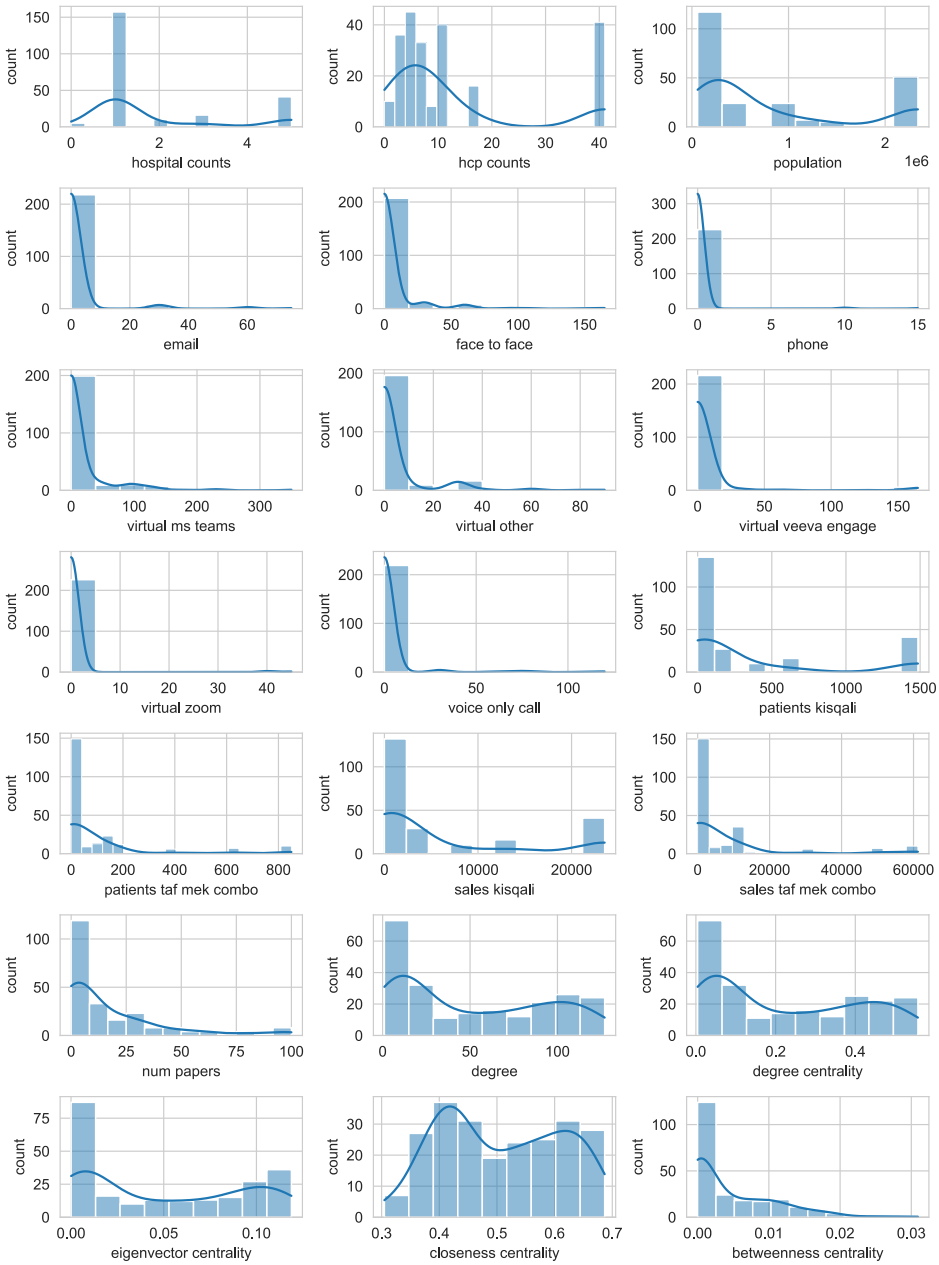


Figure 16: Distribution of metric node features

It can be clearly seen that none of the metric node features is normally or approximately normally distributed. Hence, standardization $x_{\text{stand}} = \frac{x-\mu_x}{\sigma_x}$ is not possible. We perform normalization $x_{\text{norm}} = \frac{x-x_{\min}}{x_{\max}-x_{\min}}$ instead.

After normalizing metric node features and dummy coding categorical node features, we finally obtain the node features that we use in node classification and link prediction. Table 81 shows the corresponding node feature for each index.

Index	Feature	Index	Feature
1	hospital_counts_norm	35	Karolinska Univ Sjh Solna
2	hcp_counts_norm	36	Lunds Universitet
3	population_norm	37	Länssjh Sundsvall-Härnösand
4	email_norm	38	Länssjukhuset Ryhov
5	face_to_face_norm	39	Länssjukhuset i Kalmar
6	phone_norm	40	Mälarsjukhuset
7	virtual_ms_teams_norm	41	Norrlands Universitetssjukhus
8	virtual_other_norm	42	Sahlgrenska Univ sjh
9	virtual_veeva_engage_norm	43	Skaraborgs Sjh Lidköping
10	virtual_zoom_norm	44	Skaraborgs Sjukhus Skövde
11	voice_only_call_norm	45	Skellefteå Lasarett
12	patients_kisqali_norm	46	Skånes Universitetssjukhus Lund
13	patients_taf_mek_combo_norm	47	Skånes Universitetssjukhus Malmö
14	sales_kisqali_norm	48	Sophiahemmet AB
15	sales_taf_mek_combo_norm	49	Södersjukhuset
16	num_papers_norm	50	Södra Älvsborgs Sjukhus Borås
17	degree_norm	51	Uddevalla sjukhus
18	degree_centrality_norm	52	Universitetssjukhuset Linköping
19	eigenvector_centrality_norm	53	Universitetssjukhuset Örebro
20	closeness_centrality_norm	54	Visby lasarett
21	betweenness_centrality_norm	55	Västmanlands sjukhus Västerås
22	Doctor	56	Östersunds Sjukhus
23	Nurse	57	BC
24	Akademiska sjukhuset	58	Melanoma
25	Blekingesjukhuset i Karlskrona	59	NT
26	Capio S:t Görans Sjukhus AB	60	ST
27	Centrallasarettet Växjö	61	T1
28	Centralsjukhuset i Karlstad	62	T2
29	Drottningmottagningen	63	T3
30	Falu Lasarett	64	T4
31	Gävle Sjukhus	65	S1
32	Hallands sjukhus Halmstad	66	S2
33	Hallands sjukhus Varberg	67	S3
34	Helsingborgs lasarett	68	S4

Table 81: Node features by index

In node classification, only the first 64 of the 68 node features are actually used as node features, because the last 4 node features belong to the variable "segment", the dependent variable, and are therefore used as labels. In link prediction, on the other hand, all 68 node features are used as node features.

Appendix D

We include all additional plots that have been created as part of our node classification task. By harnessing them, we can gain deeper insights into both baselines and GNNs. In detail, the following plots are provided:

- *random forest classifier*: the feature importances in each of the 10 runs for the best hyperparameter combination,
- *multilayer perceptron*: the losses, accuracies and node embeddings in each of the 10 runs for each of the two best hyperparameter combinations,
- *graph convolutional network*: the losses, accuracies and node embeddings in each of the 10 runs for the best hyperparameter combination,
- *GNN with Chebyshev convolution*: the losses, accuracies and node embeddings in each of the 10 runs for the best hyperparameter combination,
- *graph attention network*: the losses, accuracies and node embeddings in each of the 10 runs for the best hyperparameter combination.

In order to visualize the node embeddings as below, the higher-dimensional node embeddings were previously reduced to two-dimensional node embeddings using t-SNE (van der Maaten and Hinton (2008)).

Random forest classifier

Figure 17 shows the feature importances of the 64 node features in each of the 10 runs. Here the node features are indicated with indices. Table 81 shows which node feature belongs to each index.

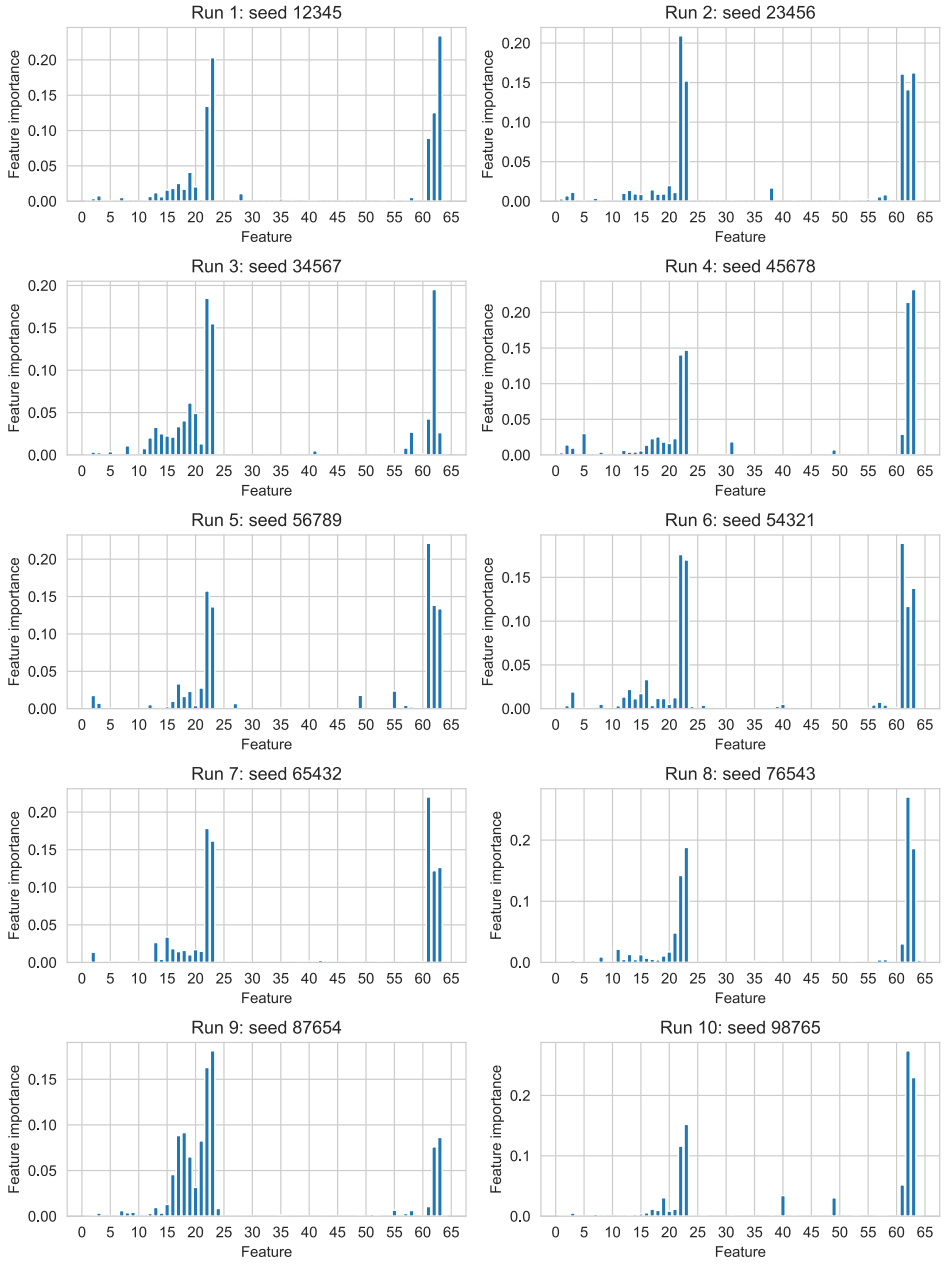


Figure 17: Random forest classifier: feature importances for best hyperparameter combination

Multilayer perceptron

The following six figures give an insight into some components of the learning process of the multilayer perceptron.

Figures 18 and 19 show for the two best hyperparameter combinations from Table 22 the training, validation and test losses in the 10 runs.

Figures 20 and 21 plot for the two best hyperparameter combinations from Table 22 the training, validation and test accuracies in the 10 runs.

Figures 22 and 23 visualize for the two best hyperparameter combinations from Table 22 the node embeddings in the 10 runs. The figures include all node embeddings at the same time, i.e., from the training, validation and test set alike.

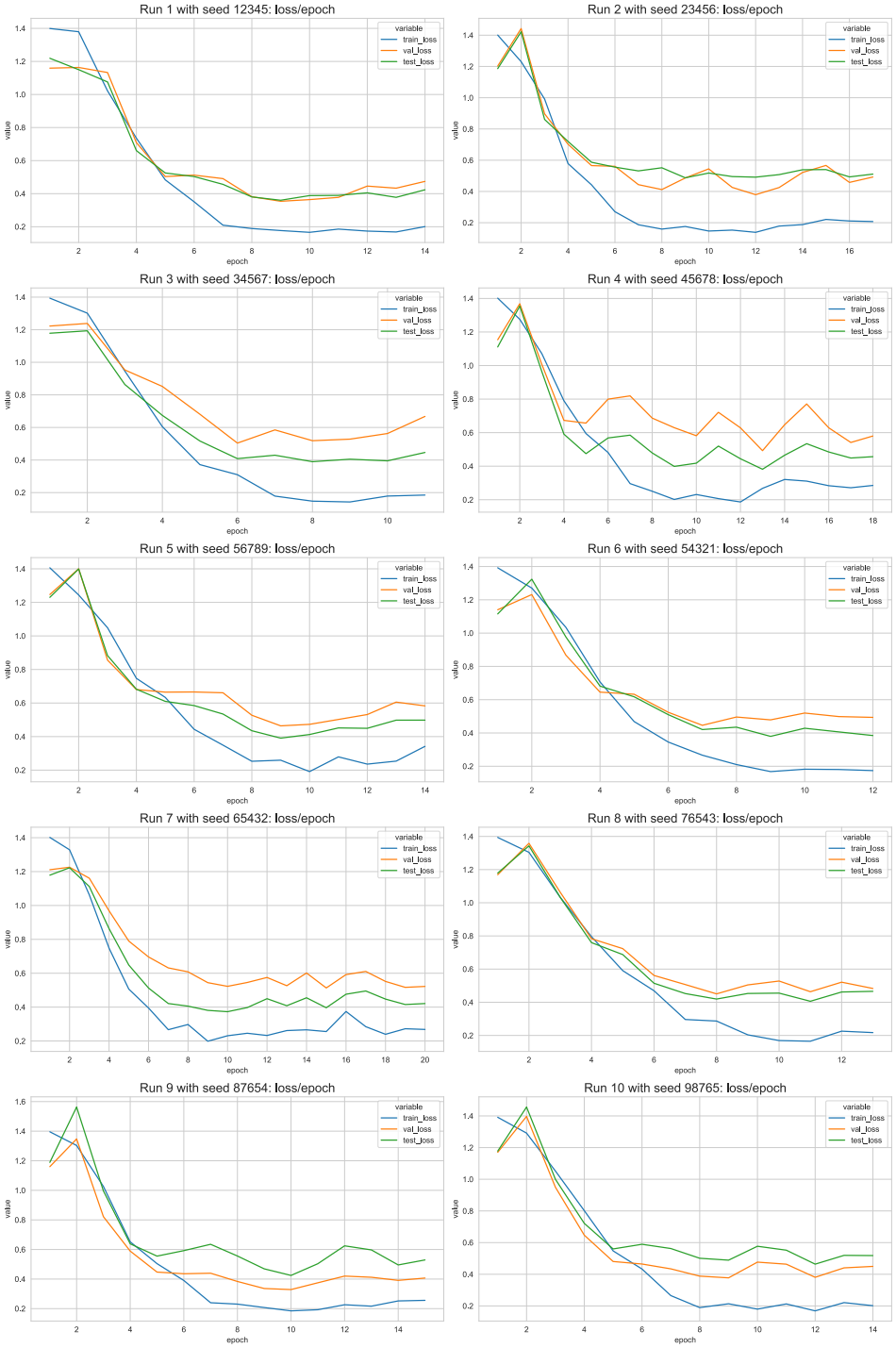


Figure 18: Multilayer perceptron: losses for best hyperparameter combination 1

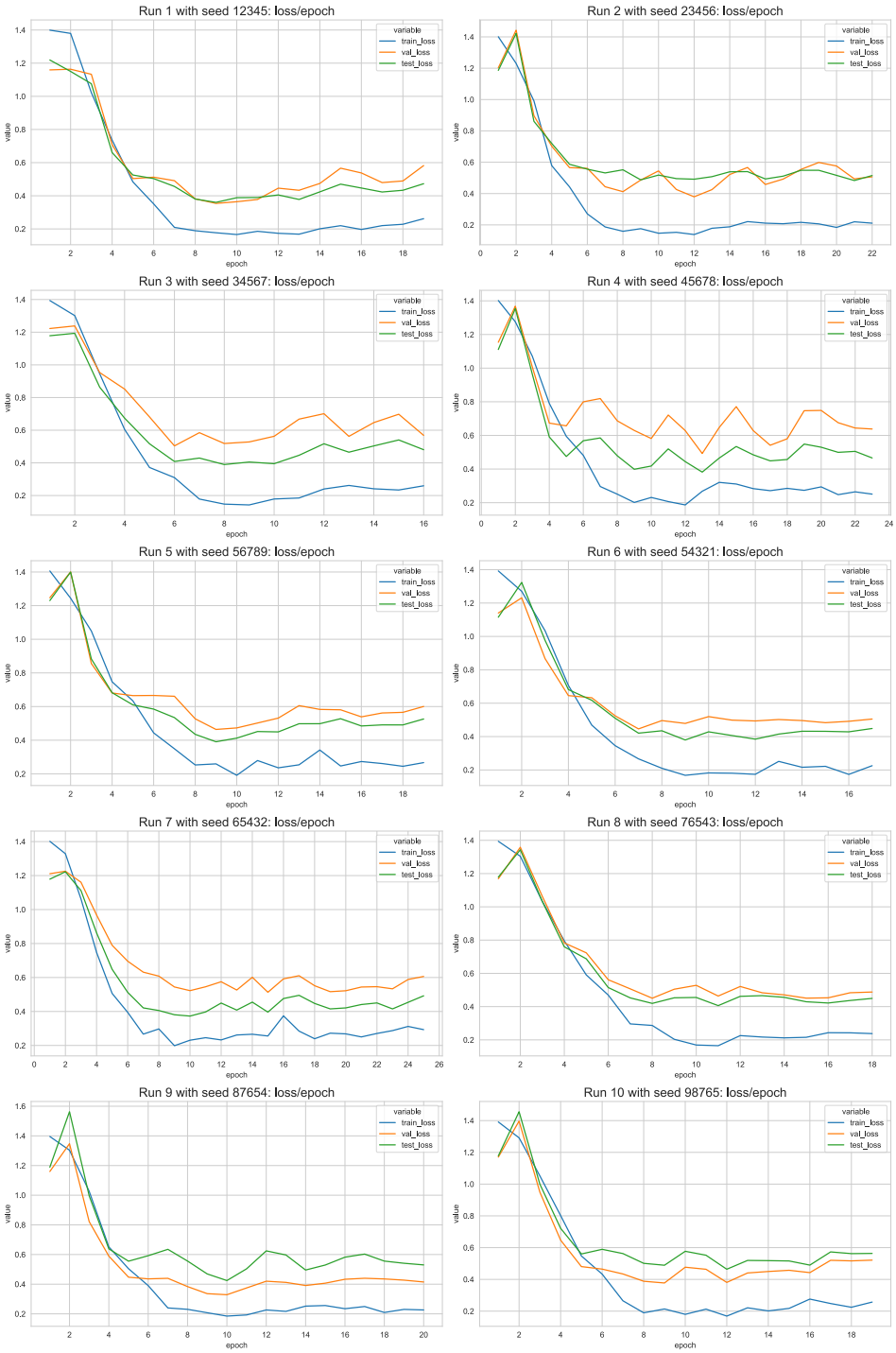


Figure 19: Multilayer perceptron: losses for best hyperparameter combination 2

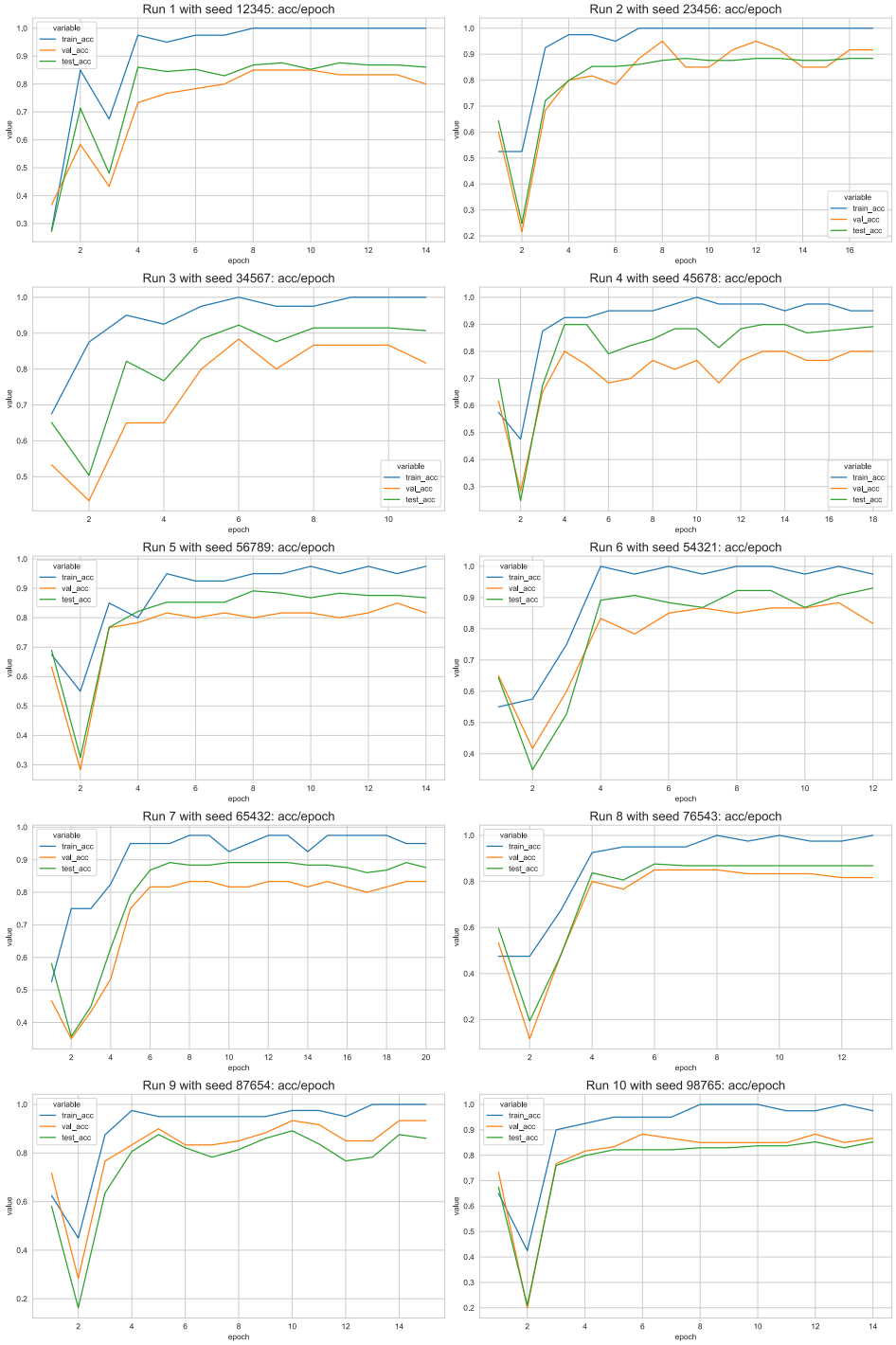


Figure 20: Multilayer perceptron: accuracies for best hyperparameter combination 1

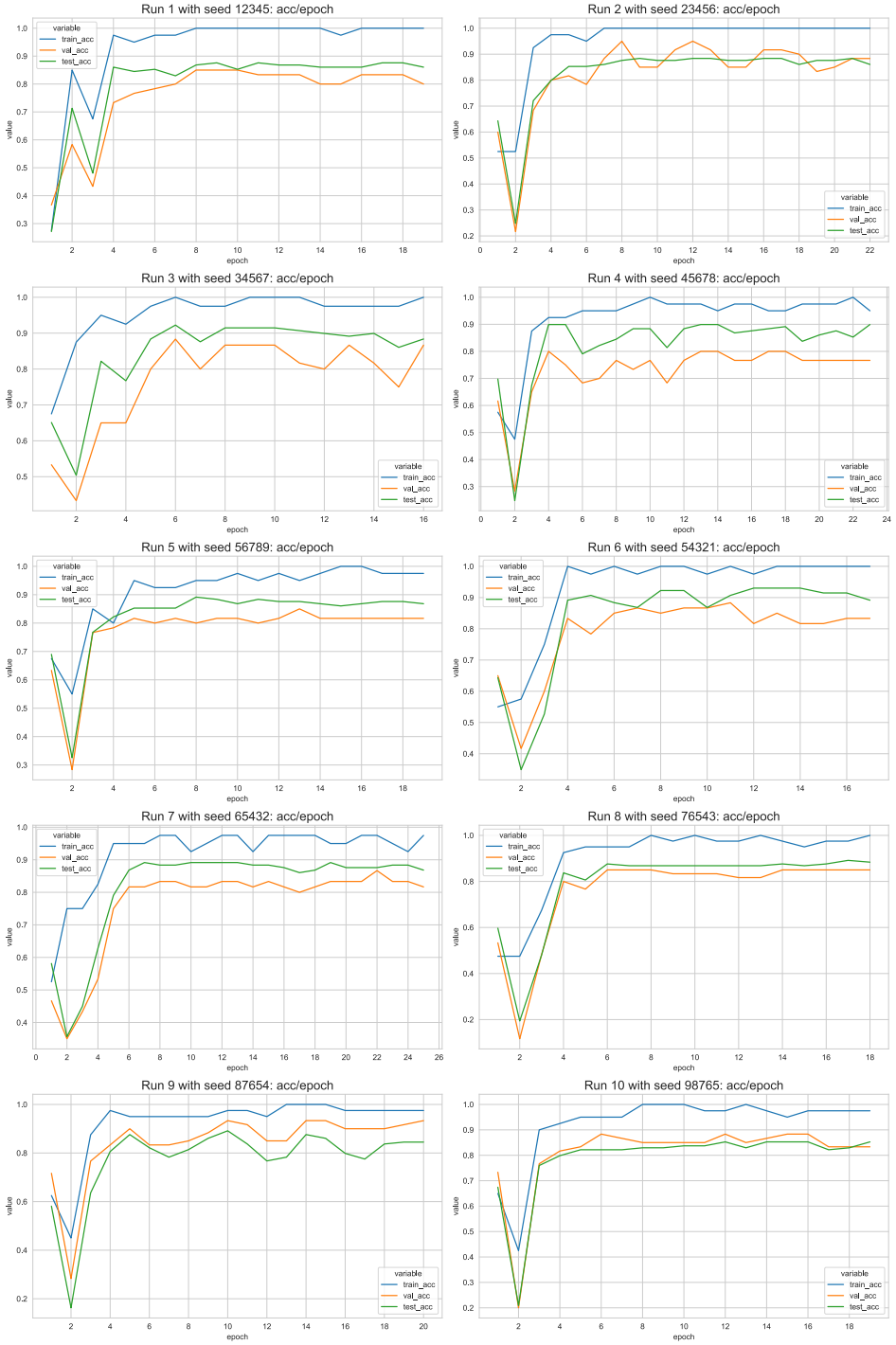


Figure 21: Multilayer perceptron: accuracies for best hyperparameter combination 2

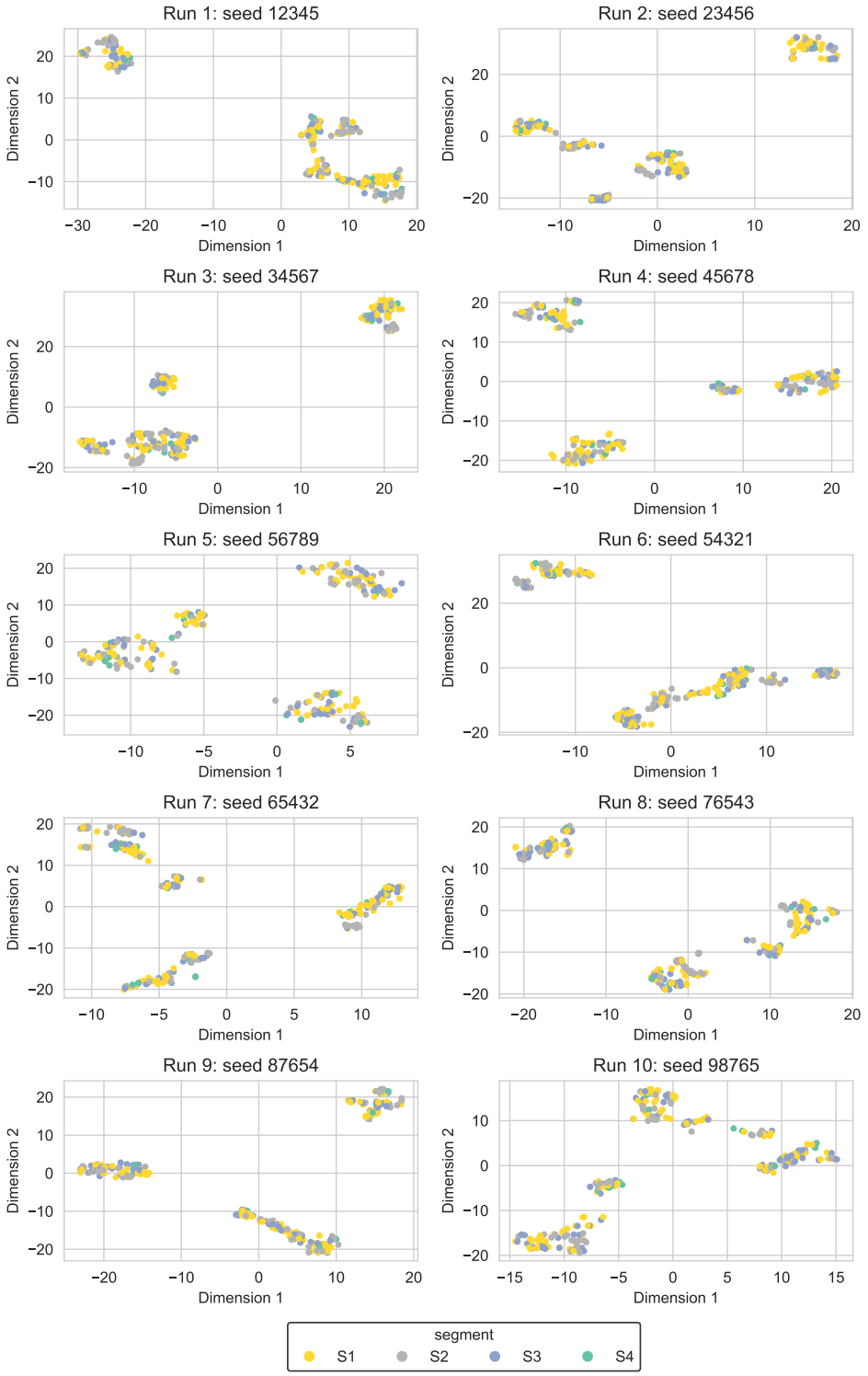


Figure 22: Multilayer perceptron: node embeddings for best hyperparameter combination 1

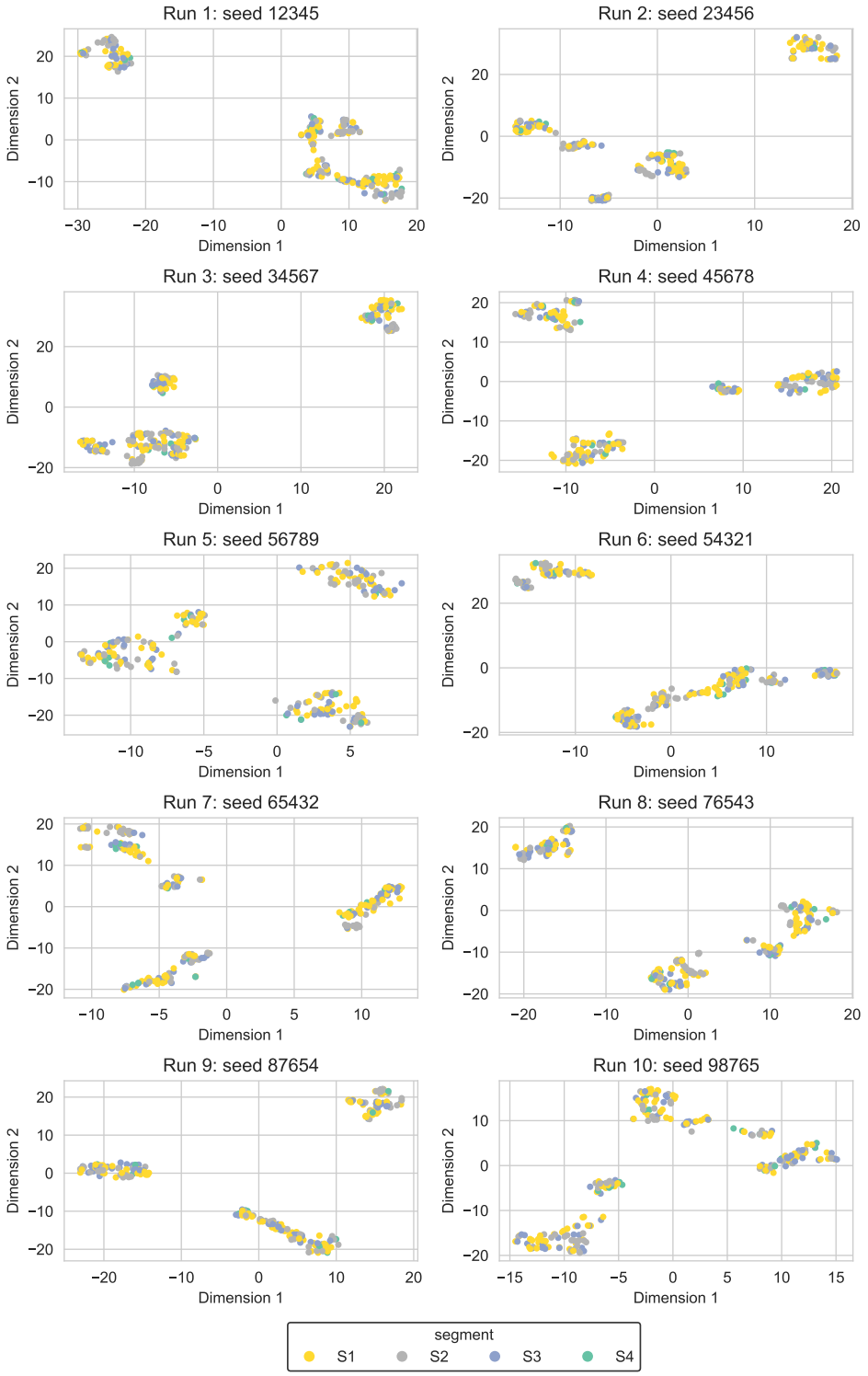


Figure 23: Multilayer perceptron: node embeddings for best hyperparameter combination 2

Graph convolutional network

The following three figures give an insight into some components of the learning process of the graph convolutional network.

Figure 24 shows for the best hyperparameter combination from Table 26 the training, validation and test losses in the 10 runs.

Figure 25 plots for the best hyperparameter combination from Table 26 the training, validation and test accuracies in the 10 runs.

Figure 26 visualizes for the best hyperparameter combination from Table 26 the node embeddings in the 10 runs. The figure includes all node embeddings at the same time, i.e., from the training, validation and test set alike.

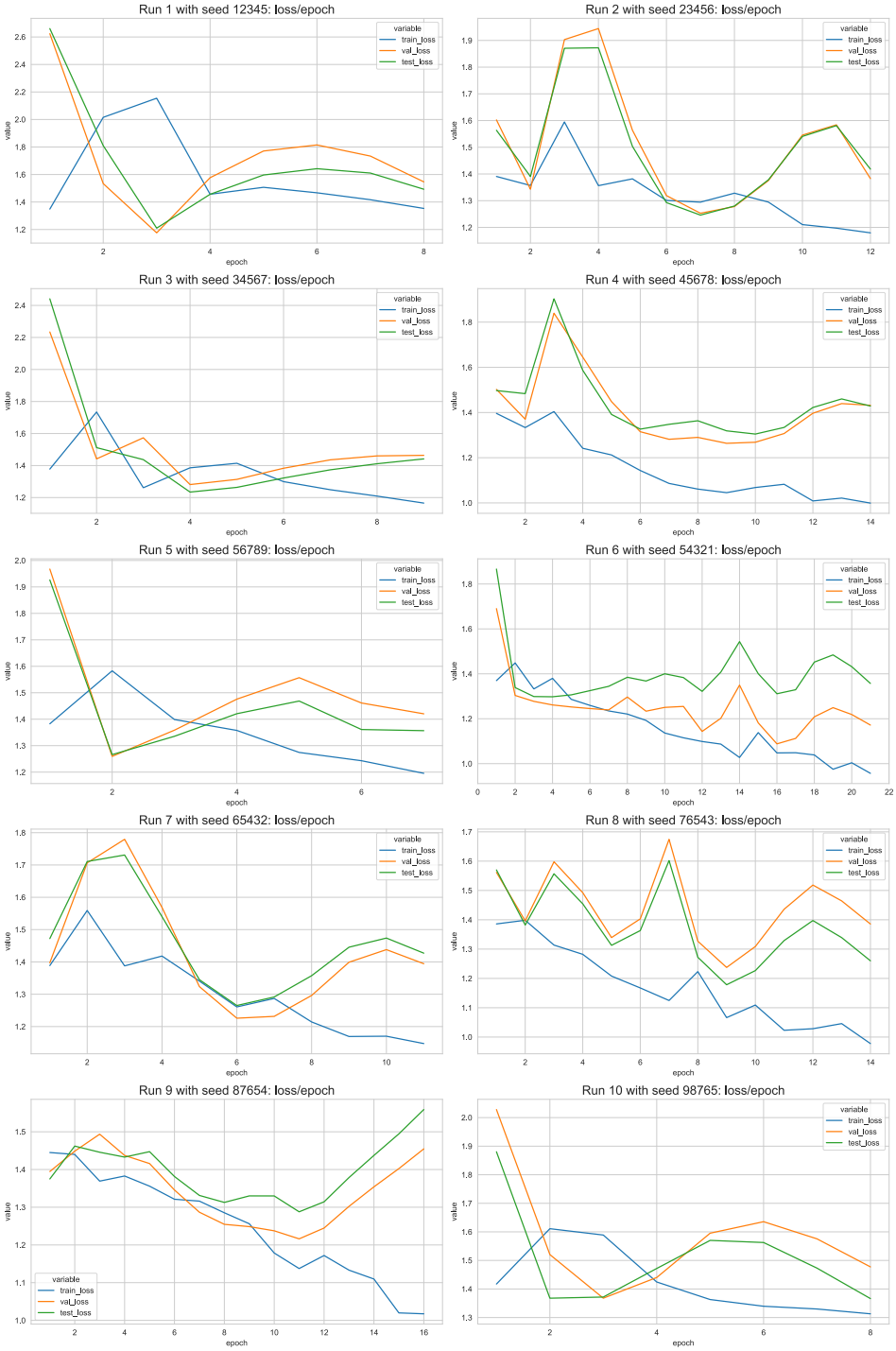


Figure 24: Graph convolutional network: losses for best hyperparameter combination

Figure 25: Graph convolutional network: accuracies for best hyperparameter combination

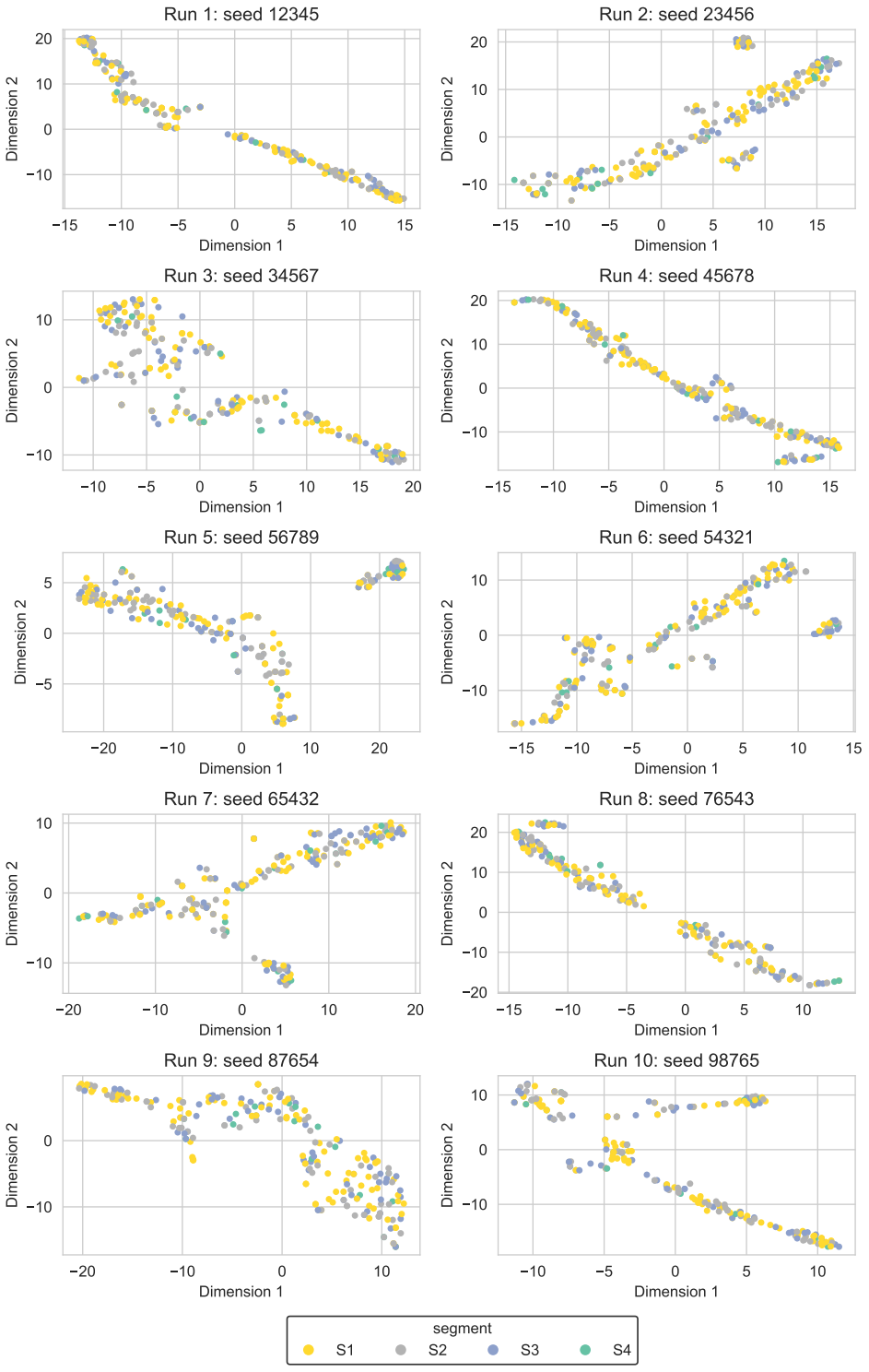


Figure 26: Graph convolutional network: node embeddings for best hyperparameter combination

GNN with Chebyshev convolution

The following three figures give an insight into some components of the learning process of the GNN with Chebyshev convolution.

Figure 27 shows for the best hyperparameter combination from Table 30 the training, validation and test losses in the 10 runs.

Figure 28 plots for the best hyperparameter combination from Table 30 the training, validation and test accuracies in the 10 runs.

Figure 29 visualizes for the best hyperparameter combination from Table 30 the node embeddings in the 10 runs. The figure includes all node embeddings at the same time, i.e., from the training, validation and test set alike.

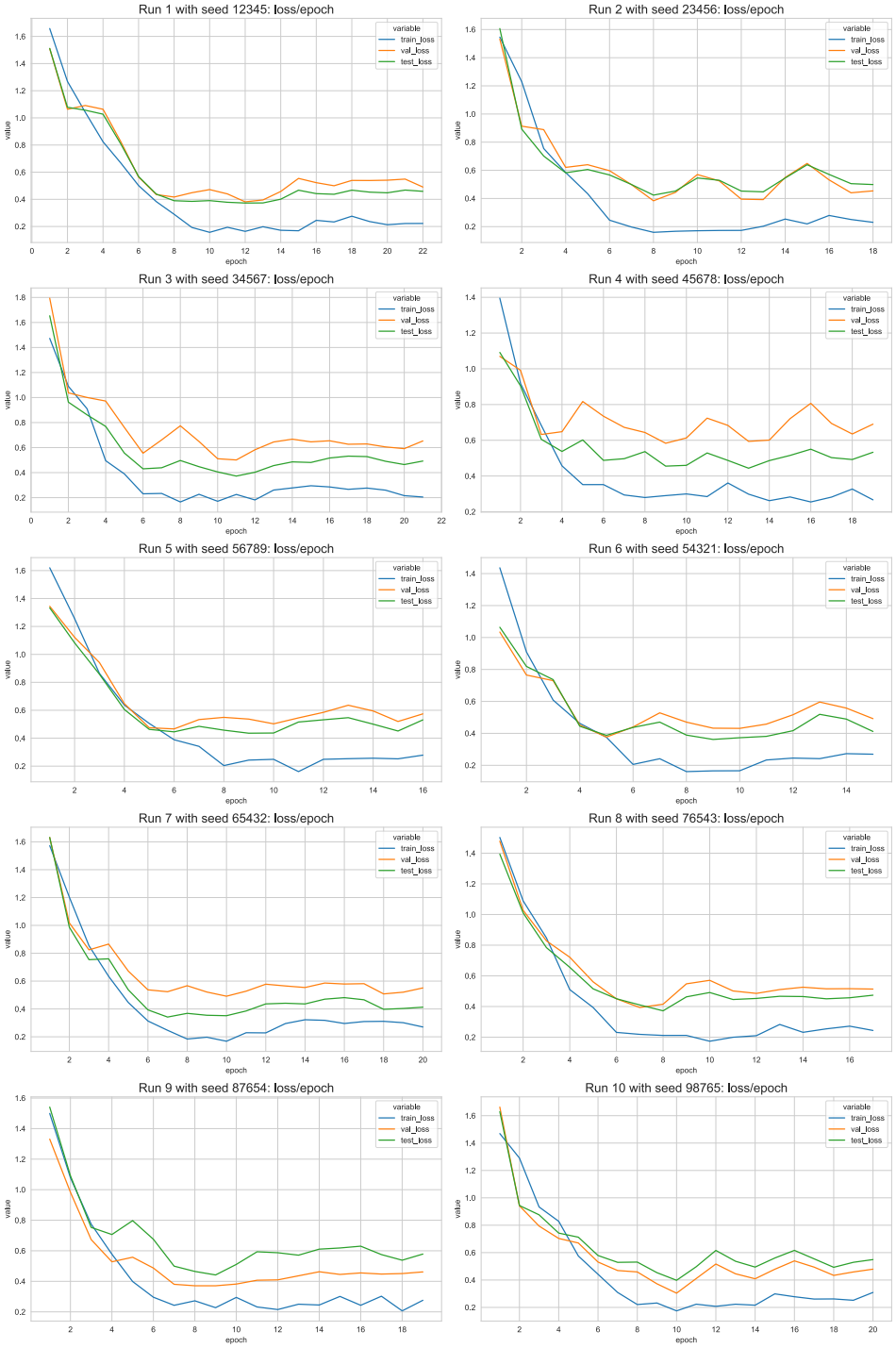


Figure 27: GNN with Chebyshev convolution: losses for best hyperparameter combination

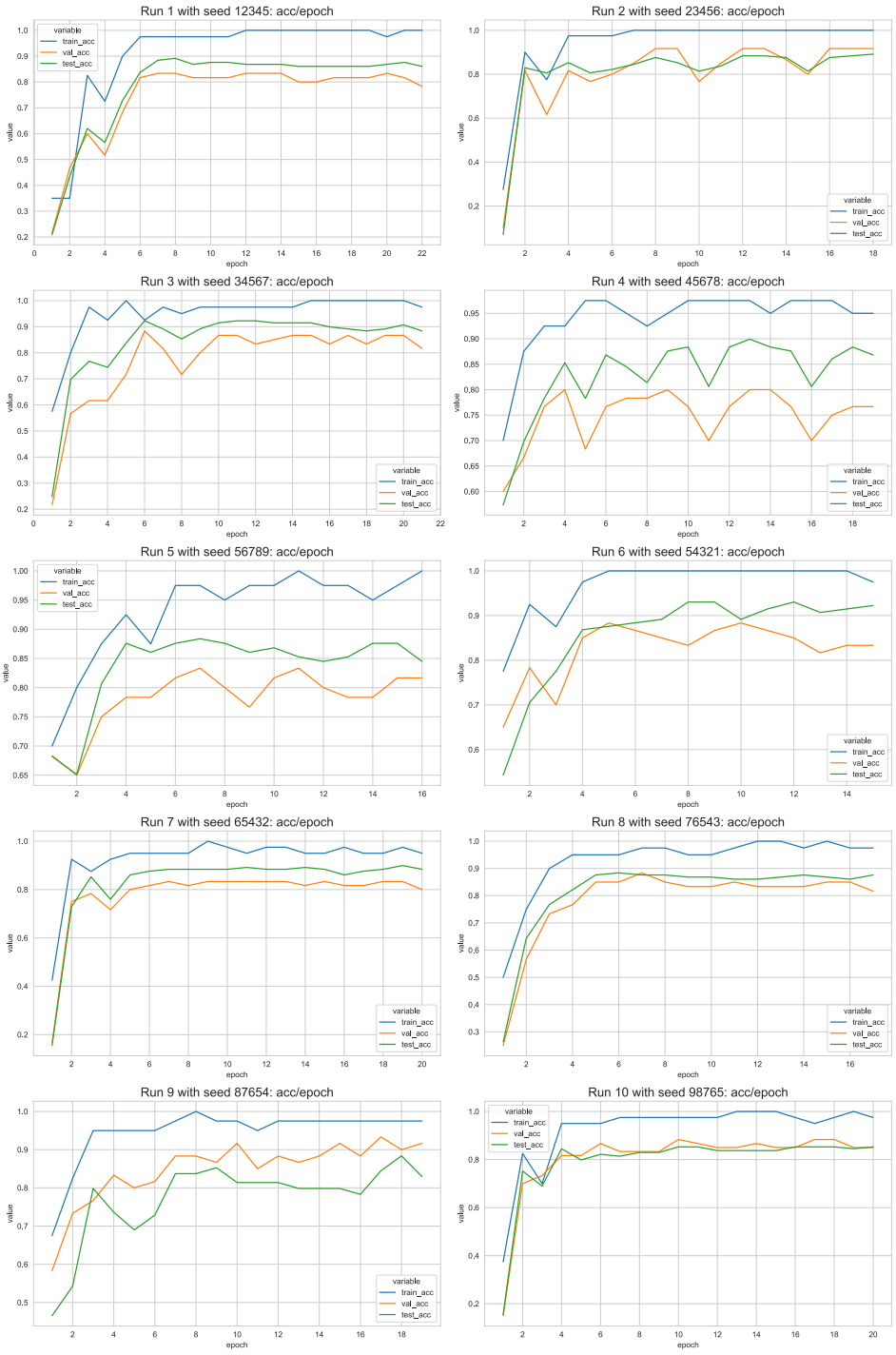


Figure 28: GNN with Chebyshev convolution: accuracies for best hyperparameter combination

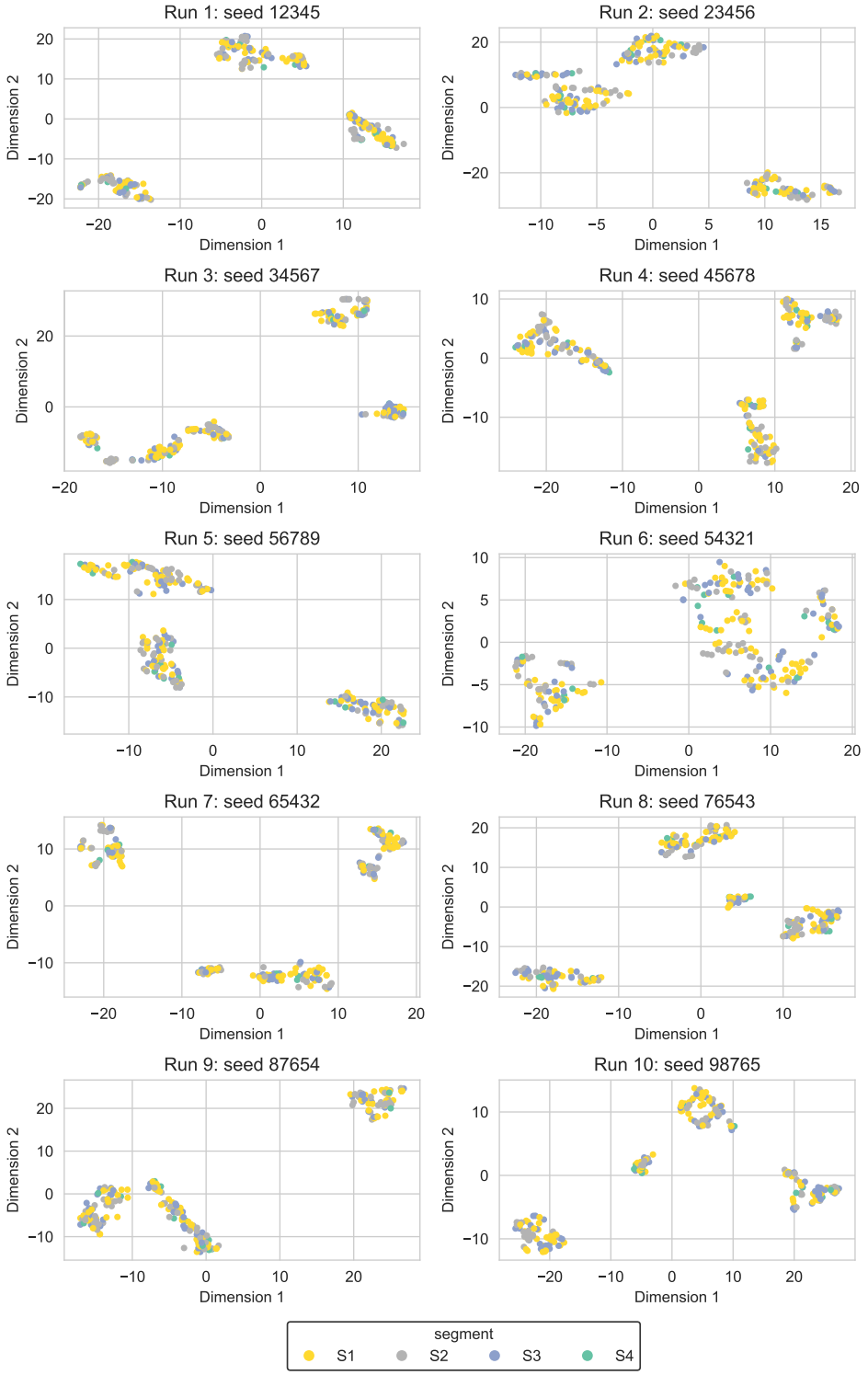


Figure 29: GNN with Chebyshev convolution: node embeddings for best hyperparameter combination

Graph attention network

The following three figures give an insight into some components of the learning process of the graph attention network.

Figure 30 shows for the best hyperparameter combination from Table 34 the training, validation and test losses in the 10 runs.

Figure 31 plots for the best hyperparameter combination from Table 34 the training, validation and test accuracies in the 10 runs.

Figure 32 visualizes for the best hyperparameter combination from Table 34 the node embeddings in the 10 runs. The figure includes all node embeddings at the same time, i.e., from the training, validation and test set alike.

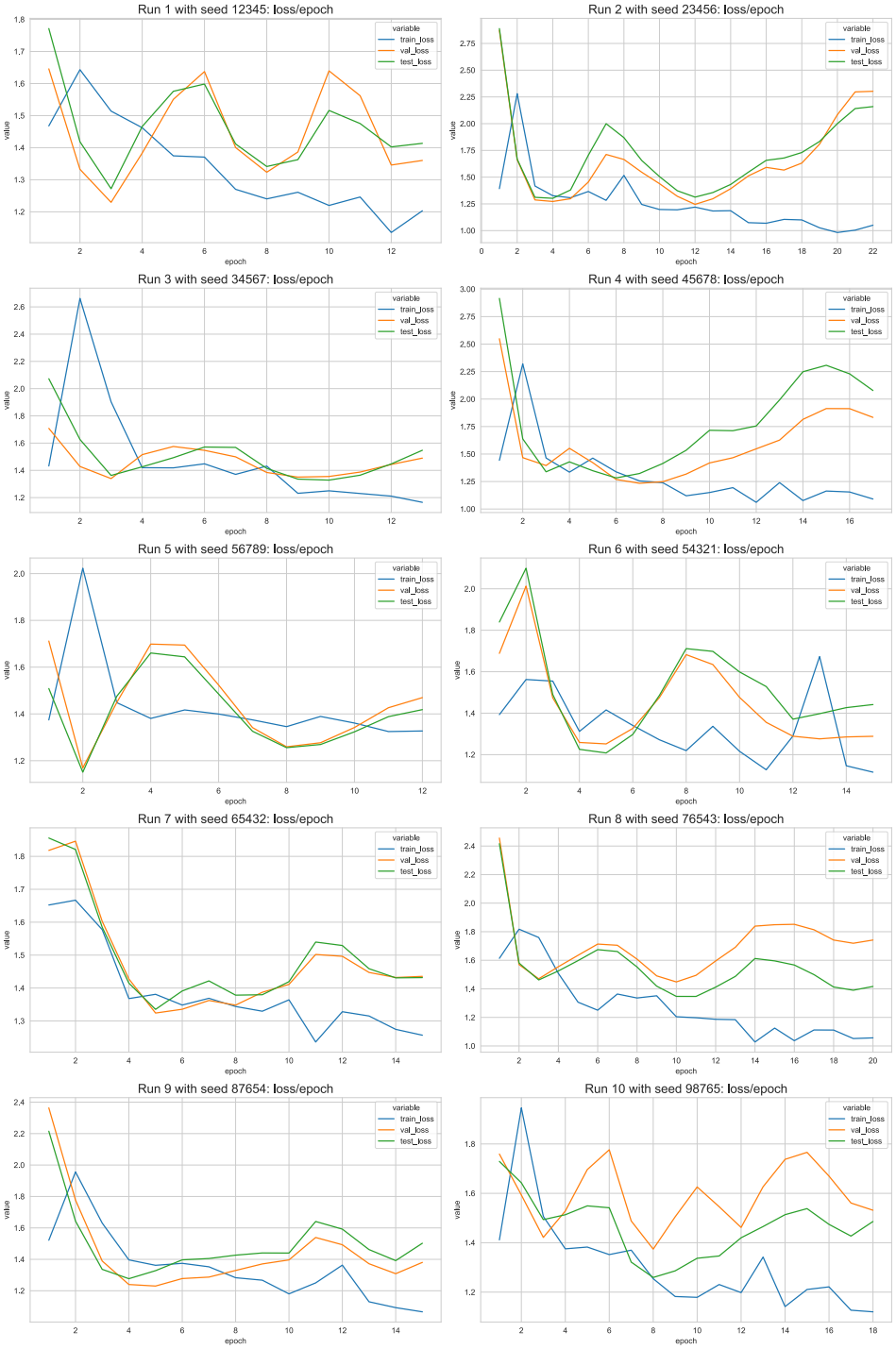


Figure 30: Graph attention network: losses for best hyperarameter combination

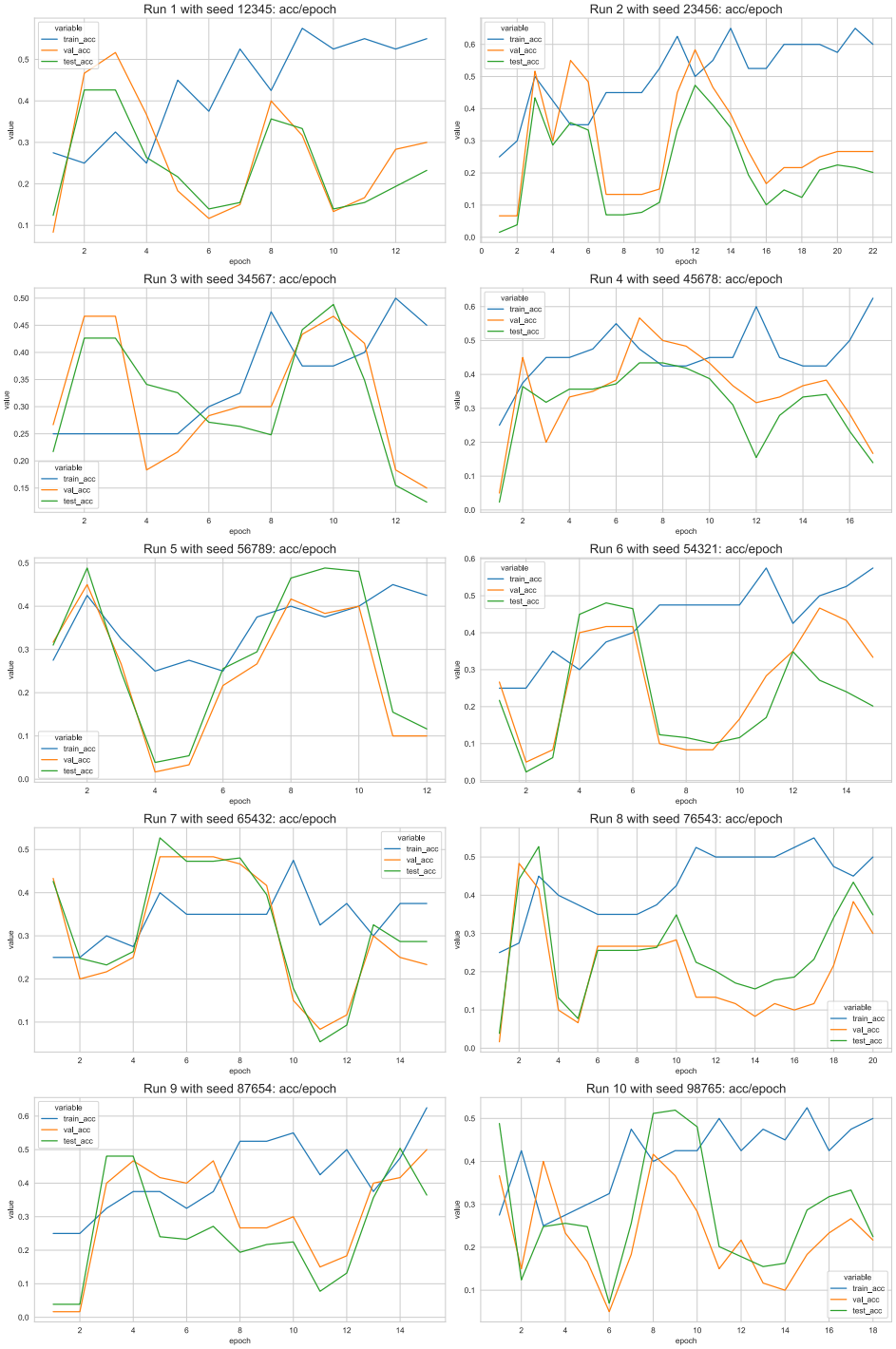


Figure 31: Graph attention network: accuracies for best hyperparameter combination

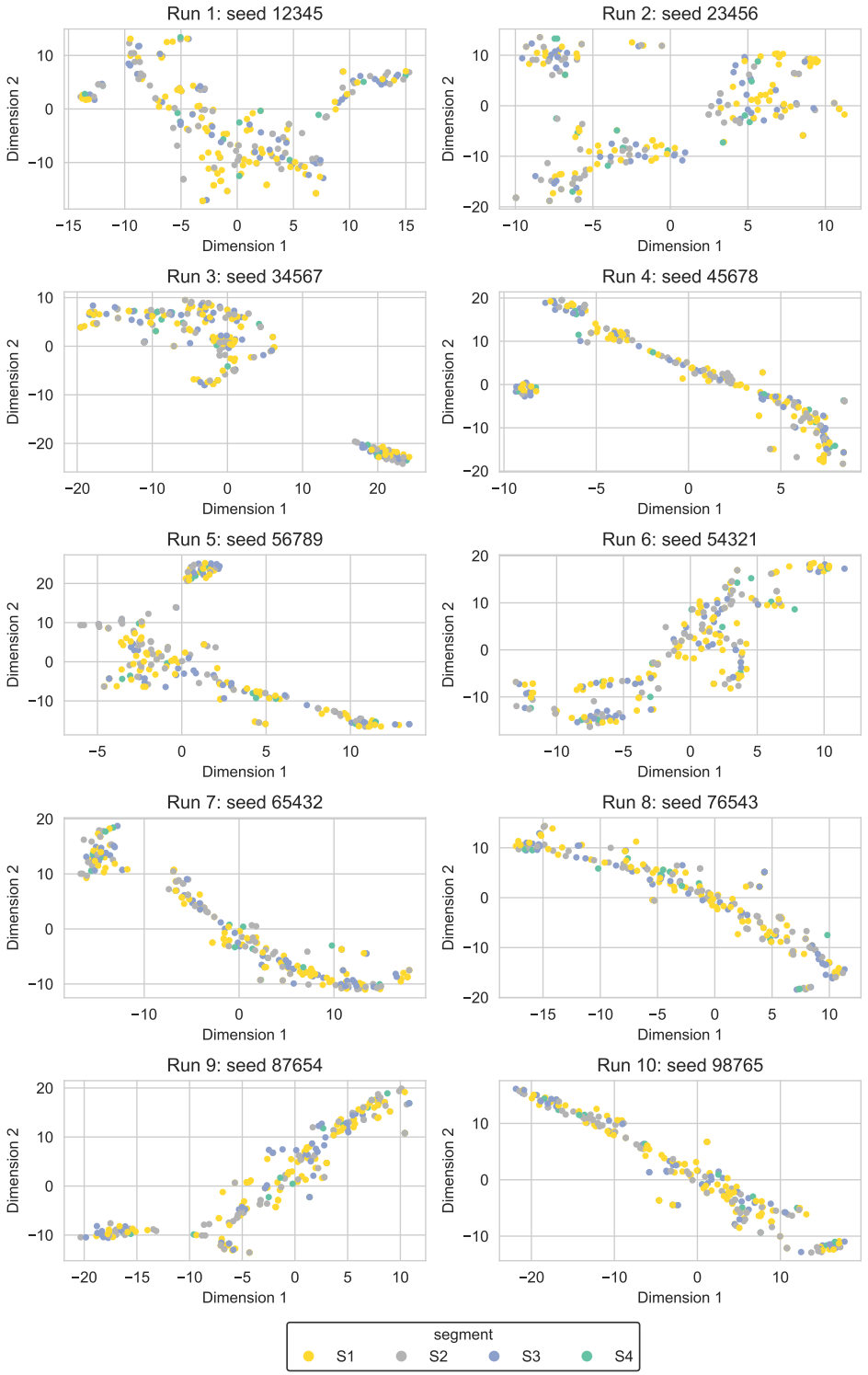


Figure 32: Graph attention network: node embeddings for best hyperparameter combination

Appendix E

We include all additional plots that have been created as part of our link prediction task. By harnessing them, we can gain deeper insights into the GNNs. In detail, the following plots are provided:

- *graph autoencoder*: the losses, average precisions and areas under the curve in each of the 10 runs for the three best hyperparameter combinations,
- *variational graph autoencoder*: the losses, average precisions and areas under the curve in each of the 10 runs for the best hyperparameter combination,
- *SEAL*: the losses, average precisions and areas under the curve in each of the 10 runs for the one hyperparameter combination for which SEAL was run.

Although link prediction performance is tracked using both the average precision and the area under the curve, the best hyperparameter combination is determined exclusively on the basis of the average precision.

Graph autoencoder

The following three figures give an insight into some components of the learning process of the graph autoencoder.

Figure 33 shows for the three best hyperparameter combinations from Table 56 the training, validation and test losses in the 10 runs.

Figure 34 plots for the three best hyperparameter combinations from Table 56 the training, validation and test average precisions in the 10 runs.

Figure 35 visualizes for the three best hyperparameter combinations from Table 56 the training, validation and test areas under the curve in the 10 runs.

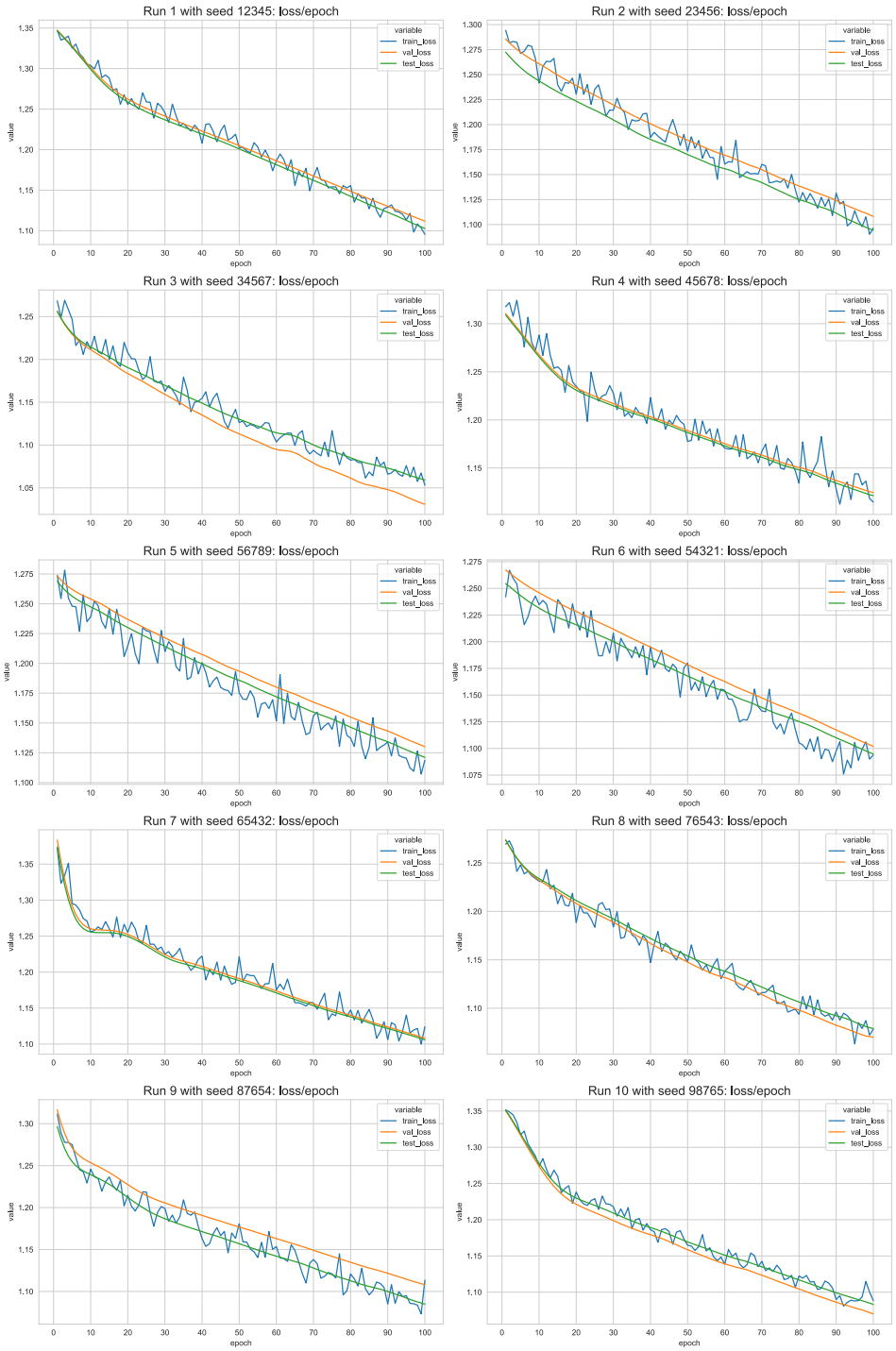


Figure 33: Graph autoencoder: losses for three best hyperparameter combinations

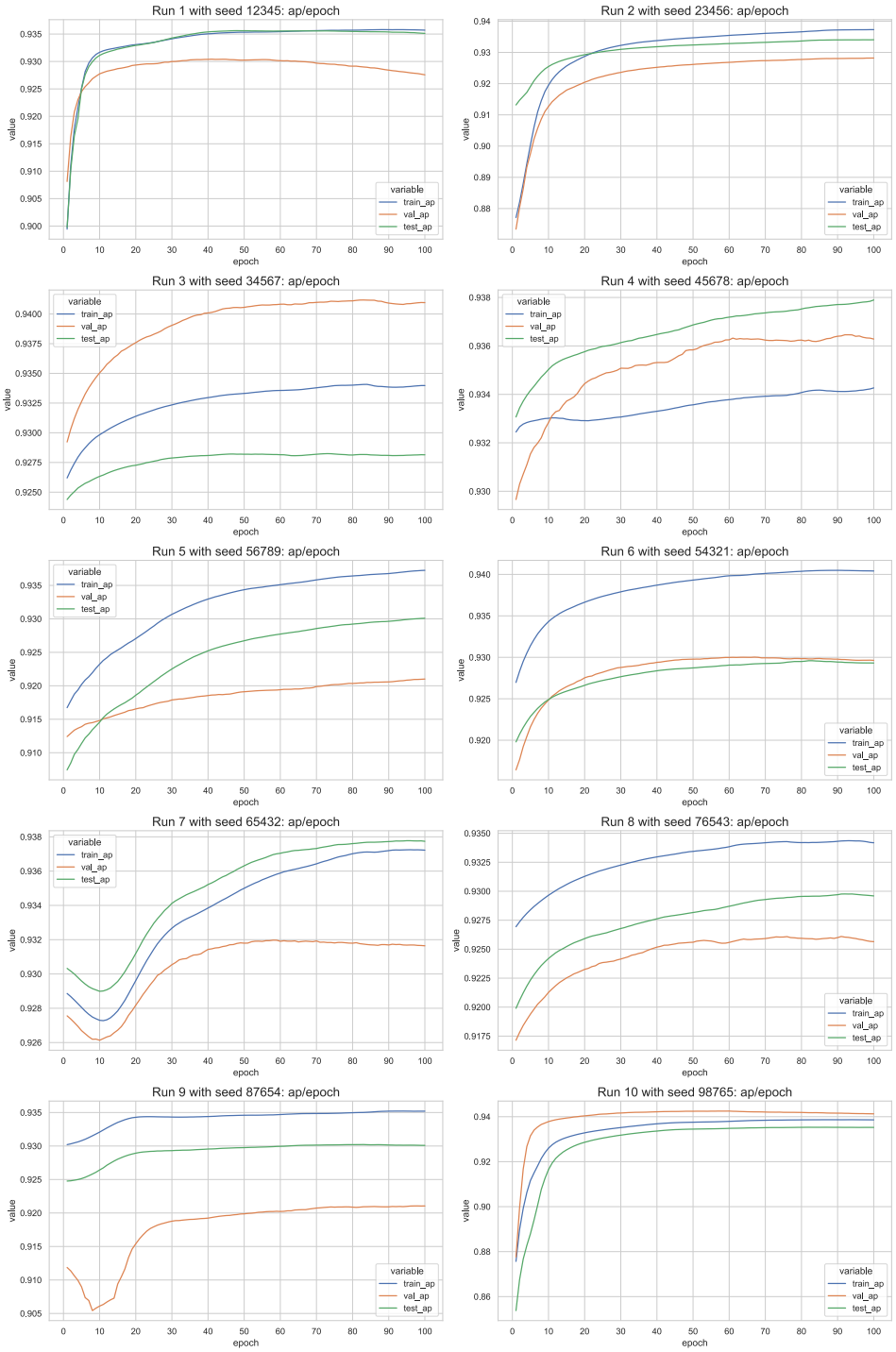


Figure 34: Graph autoencoder: average precisions for three best hyperparameter combinations

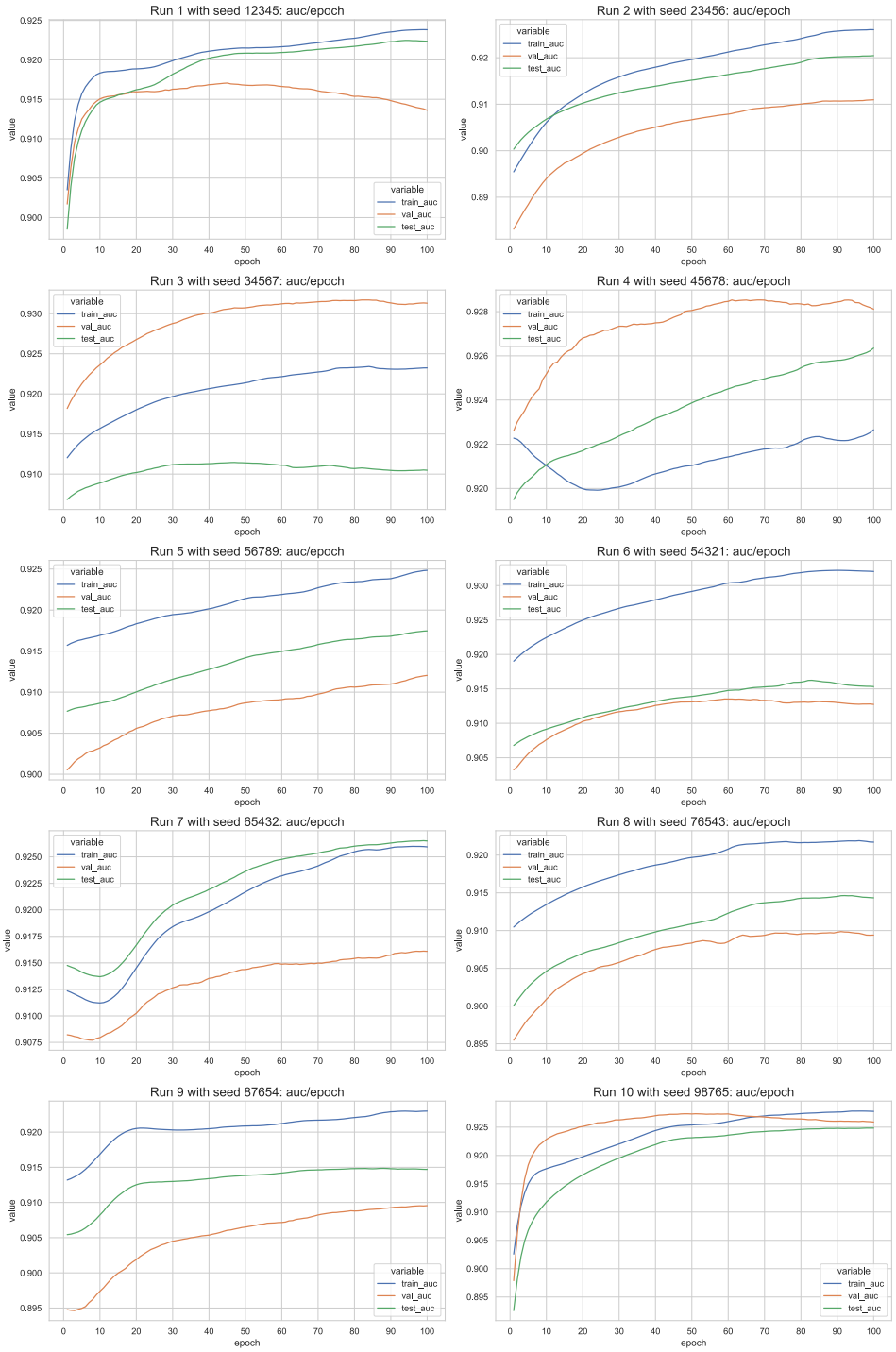


Figure 35: Graph autoencoder: areas under the curve for three best hyperparameter combinations

Variational graph autoencoder

The following three figures give an insight into some components of the learning process of the variational graph autoencoder.

Figure 36 shows for the best hyperparameter combination from Table 60 the training, validation and test losses in the 10 runs.

Figure 37 plots for the best hyperparameter combination from Table 60 the training, validation and test average precisions in the 10 runs.

Figure 38 visualizes for the best hyperparameter combination from Table 60 the training, validation and test areas under the curve in the 10 runs.

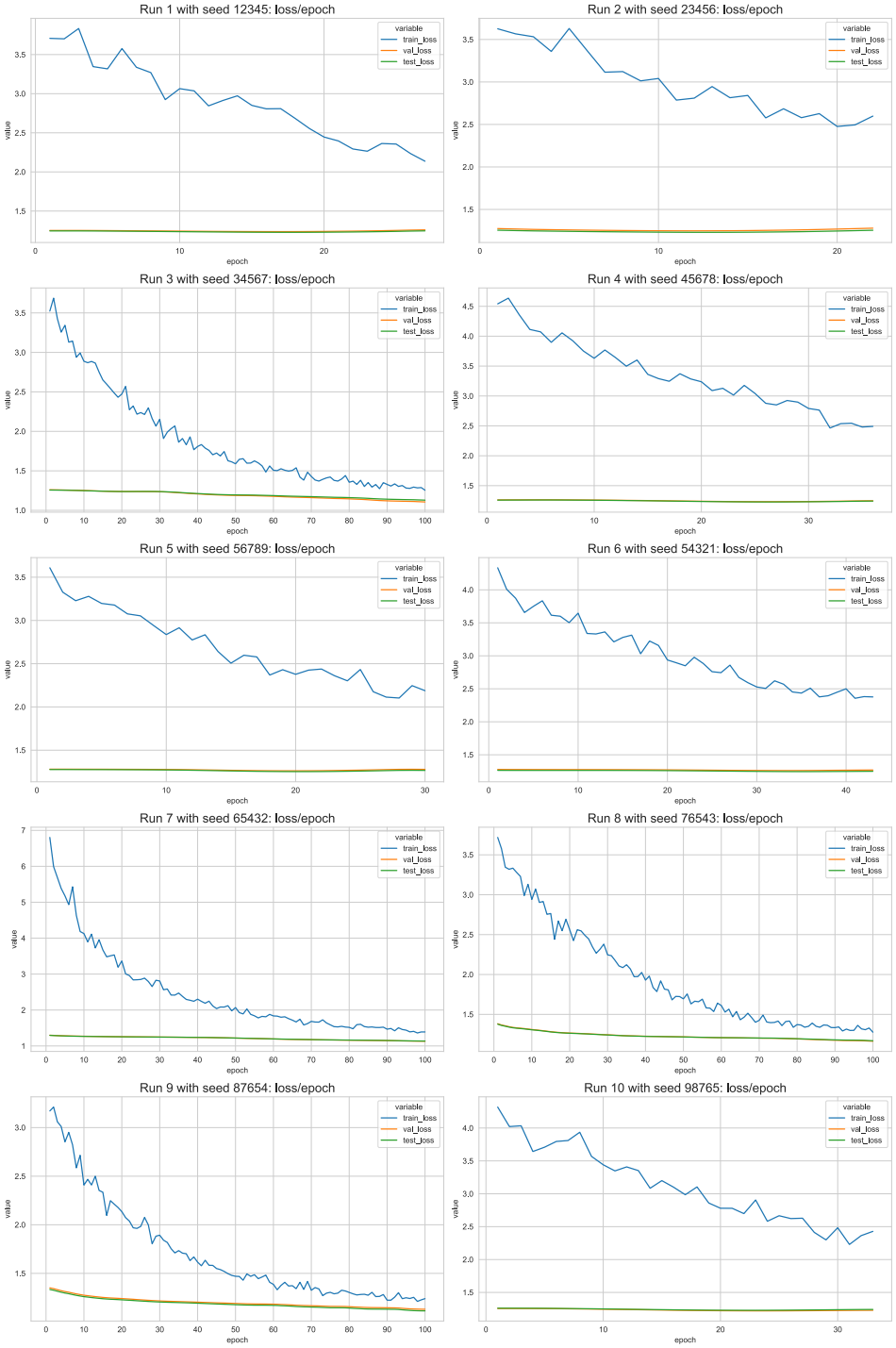


Figure 36: Variational graph autoencoder: losses for best hyperparameter combination

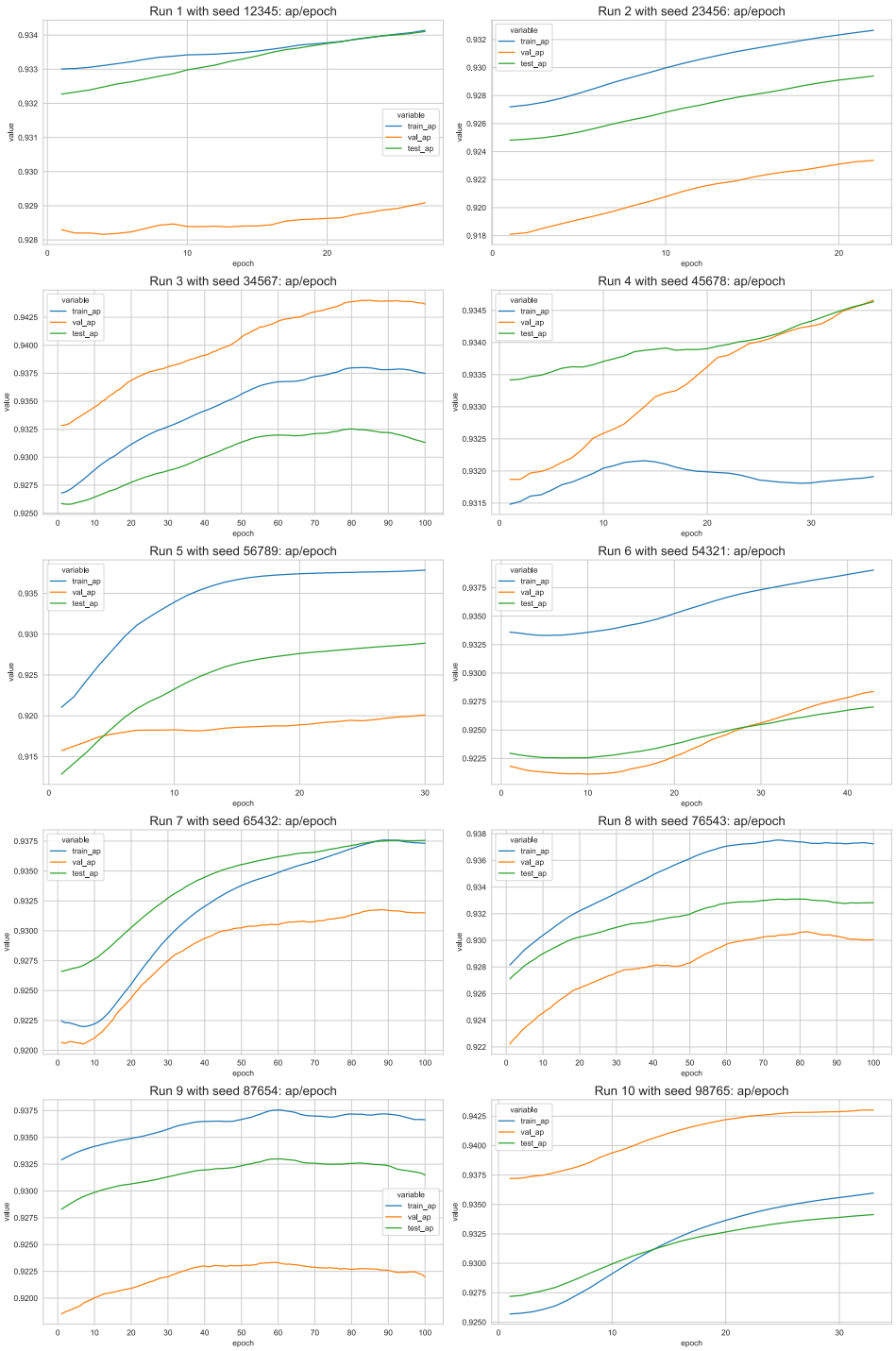


Figure 37: Variational graph autoencoder: average precisions for best hyperparameter combination

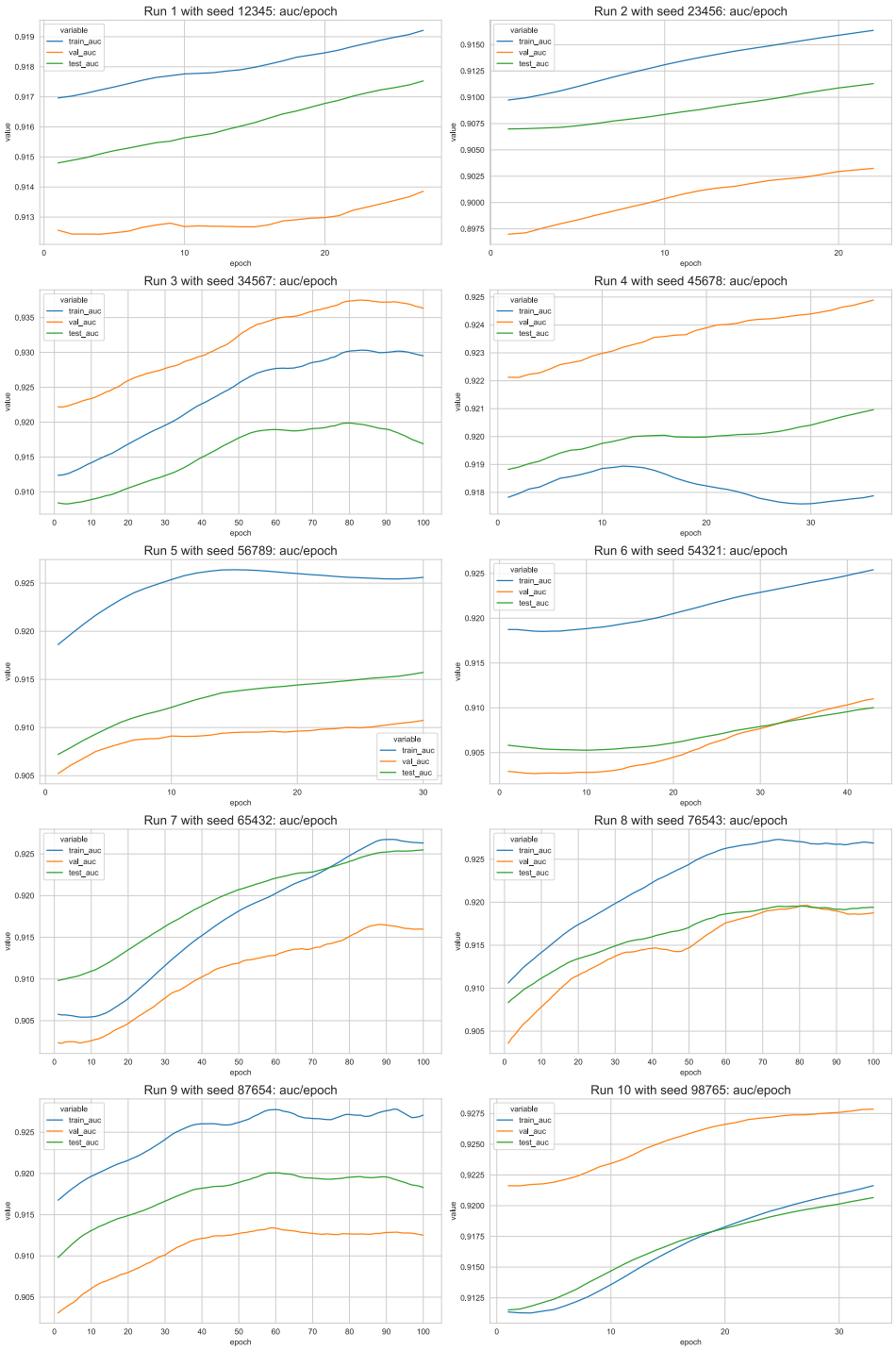


Figure 38: Variational graph autoencoder: areas under the curve for best hyperparameter combination

SEAL

The following three figures give an insight into some components of the learning process of SEAL.

Figure 39 shows for the hyperparameter combination from Table 63 the training, validation and test losses in the 10 runs.

Figure 40 plots for the hyperparameter combination from Table 63 the training, validation and test average precisions in the 10 runs.

Figure 41 visualizes for the hyperparameter combination from Table 63 the training, validation and test areas under the curve in the 10 runs.

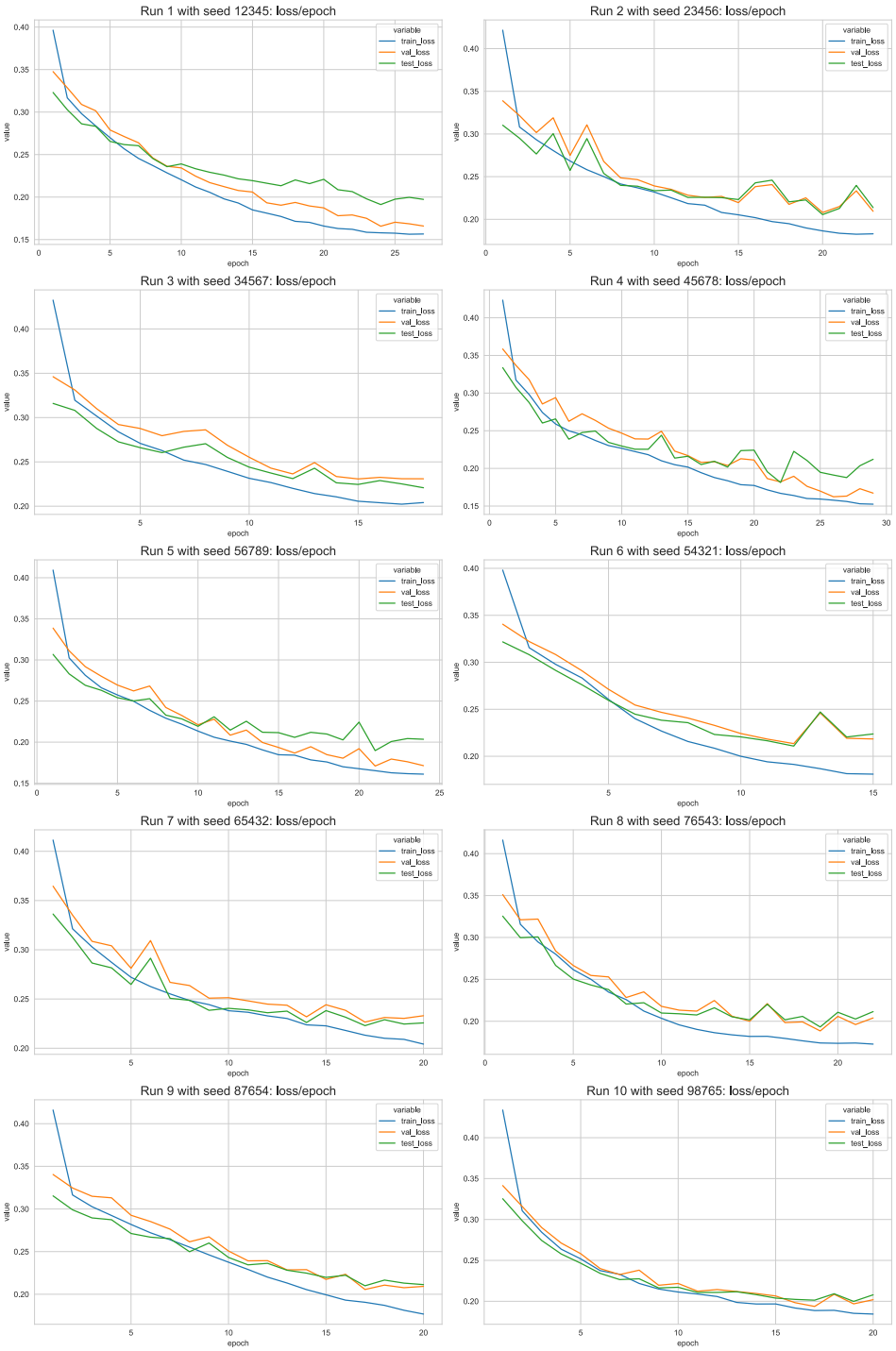


Figure 39: SEAL: losses for hyperparameter combination chosen

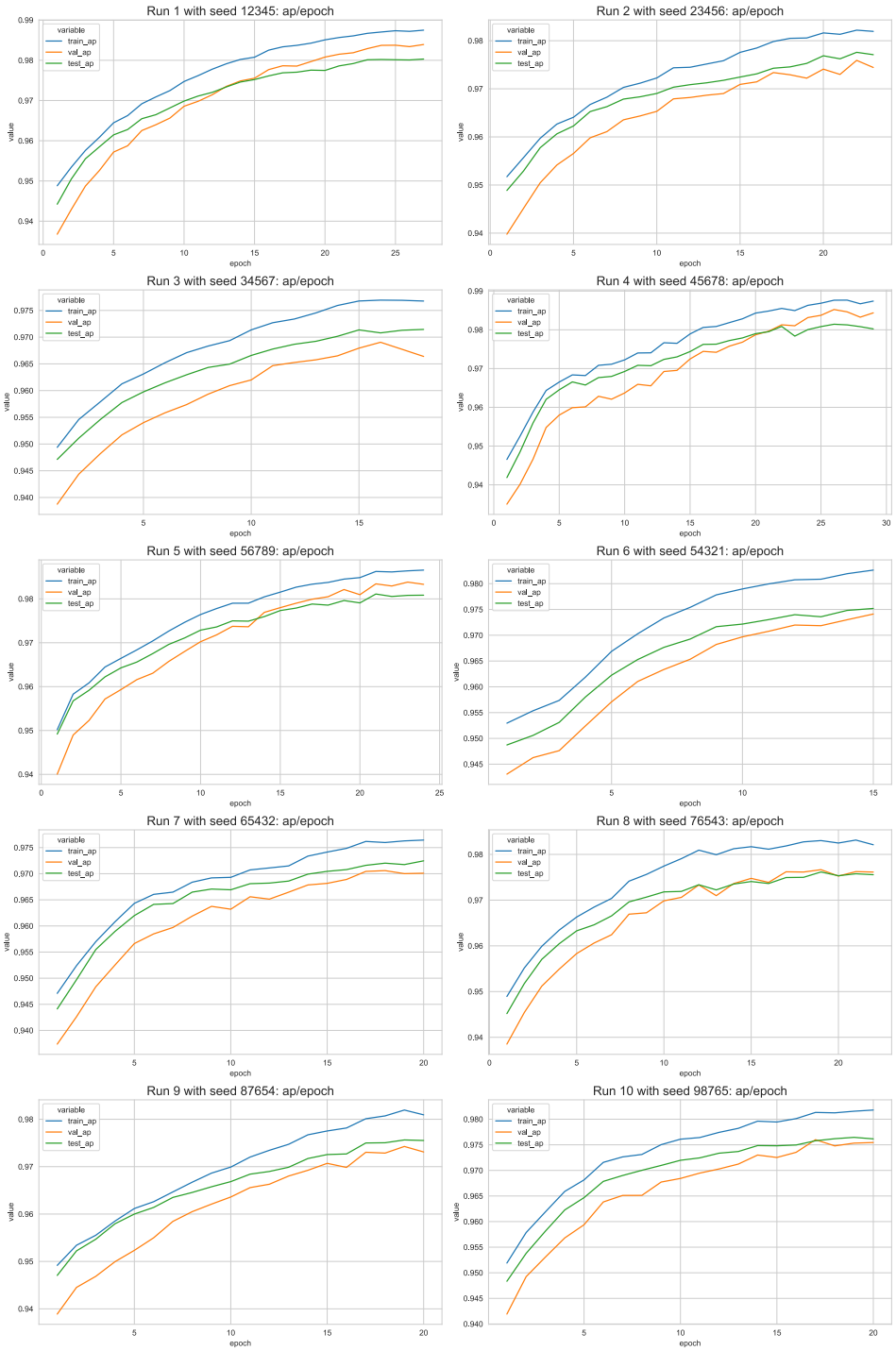


Figure 40: SEAL: average precisions for hyperparameter combination chosen

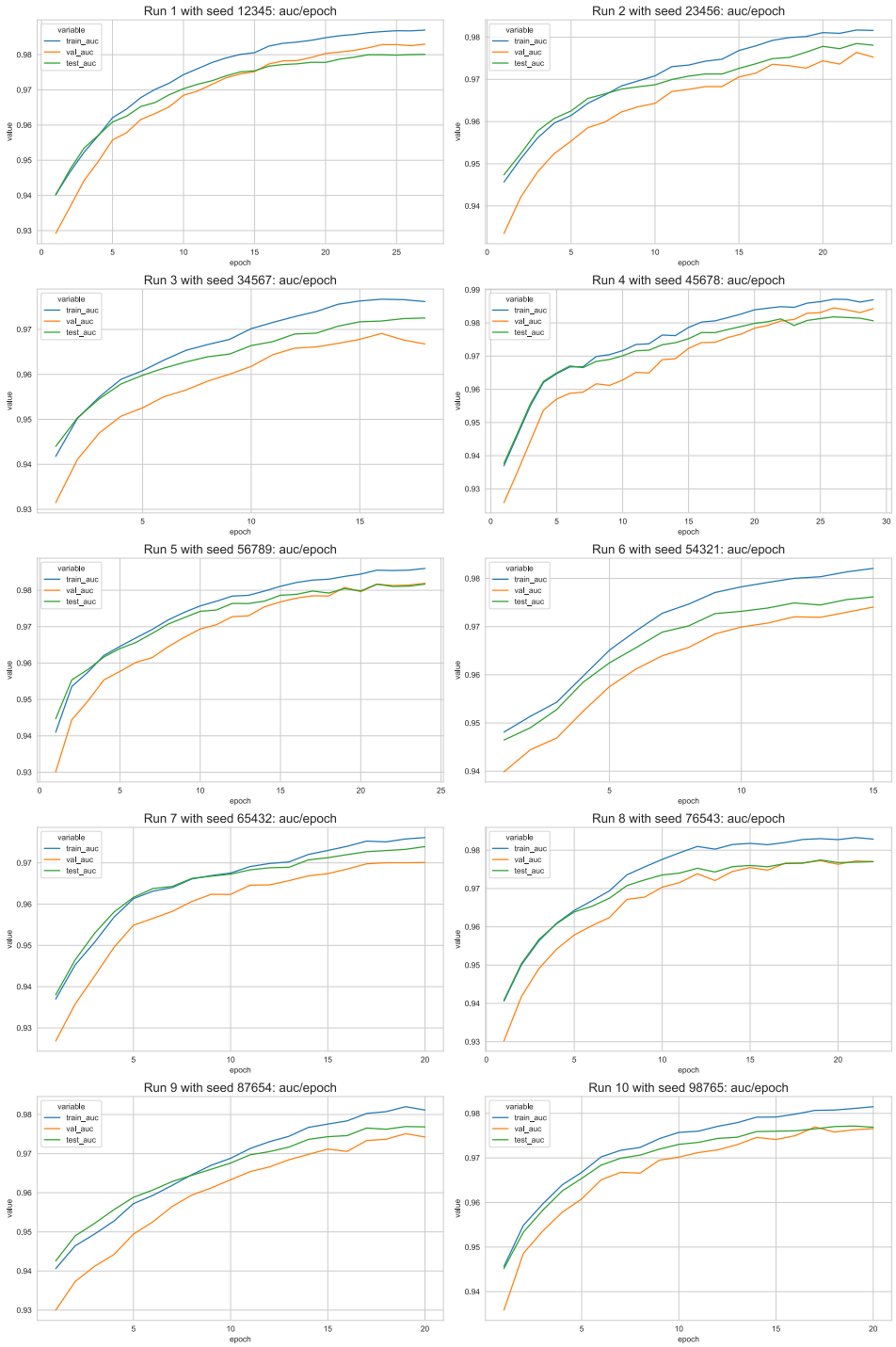


Figure 41: SEAL: areas under the curve for hyperparameter combination chosen

Appendix F

We include all plots that have been created as part of our time series forecasting task. By harnessing them, we can gain deeper insights into GTS. In detail, the following plots are provided:

- *BC forecasts*: the original sales of Kisqali and the sales forecasts obtained using static forecasting, historical average forecasting, vector autoregression and GTS in each of the BC regions for the three best hyperparameter combinations,
- *melanoma forecasts*: the original sales of Tafinlar-Mekinist and the sales forecasts obtained using static forecasting, historical average forecasting, vector autoregression and GTS in each of the melanoma regions for the three best hyperparameter combinations,
- *losses*: the losses for the three best hyperparameter combinations,
- *metrics*: the MAEs, MAPEs and RMSEs for the three best hyperparameter combinations,
- *a priori graph*: the a priori graph used by GTS for the three best hyperparameter combinations, along with a table summarizing it,
- *learned graph*: the graph learned by GTS for the three best hyperparameter combinations, together with a table with a summary of it.

Although forecasting performance is tracked using the MAE, MAPE and RMSE, the best hyperparameter combination is determined exclusively on the basis of the MAE.

BC forecasts

The following six figures show the original Kisqali sales from 09-2019 to 08-2021 in blue and the Kisqali sales forecasts for all 35 BC regions. The static forecasts are orange, the historical average forecasts are green, the vector autoregression forecasts are red and the GTS forecasts are purple.

We have not generated forecasts for the sales in the training set, therefore there are no forecasts before 12-2020. The forecasts from 12-2020 to 02-2021 belong to the validation set and the forecasts from 03-2021 to 08-2021 to the test set.

Recall that the BC regions are those regions that do not bear the suffix "ONCO".

Figure 42 plots the original Kisqali sales from 09-2019 to 08-2021 and the Kisqali sales forecasts from 12-2020 to 08-2021 for BC regions: Blekinge, Dalarna, Gävleborg-Gävle, Halland-Halmstad, Halland-Varberg-Falkenberg, Jämtland.

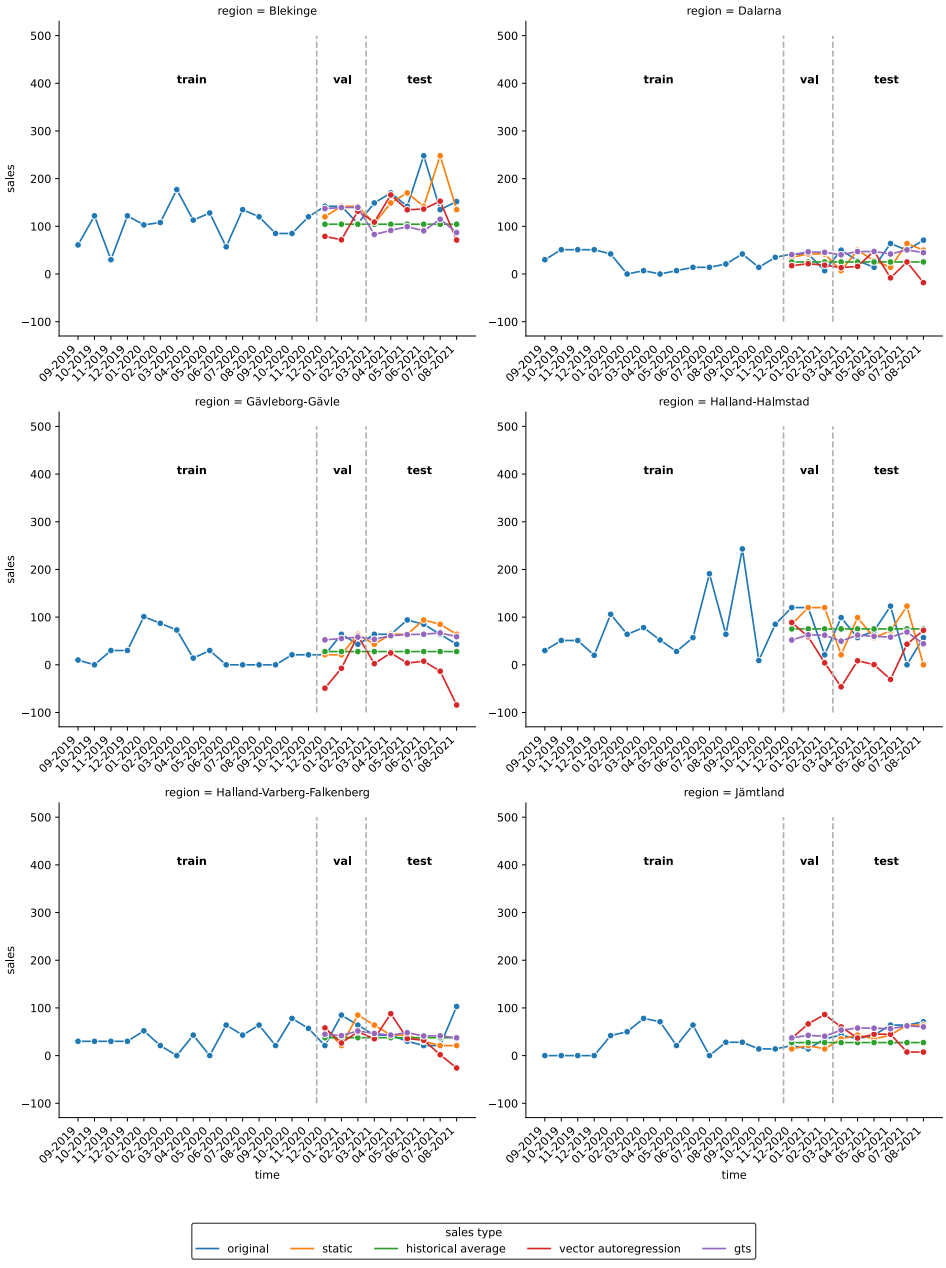


Figure 42: Sales forecasts for BC regions 1-6

Figure 43 plots the original Kisqali sales from 09-2019 to 08-2021 and the Kisqali sales forecasts from 12-2020 to 08-2021 for BC regions: Jönköping-Jönköping, Jönköping-Nässjö-Eksjö, Jönköping-Värnamo, Kalmar, Kronoberg-Ljungby, Kronoberg-Växjö.

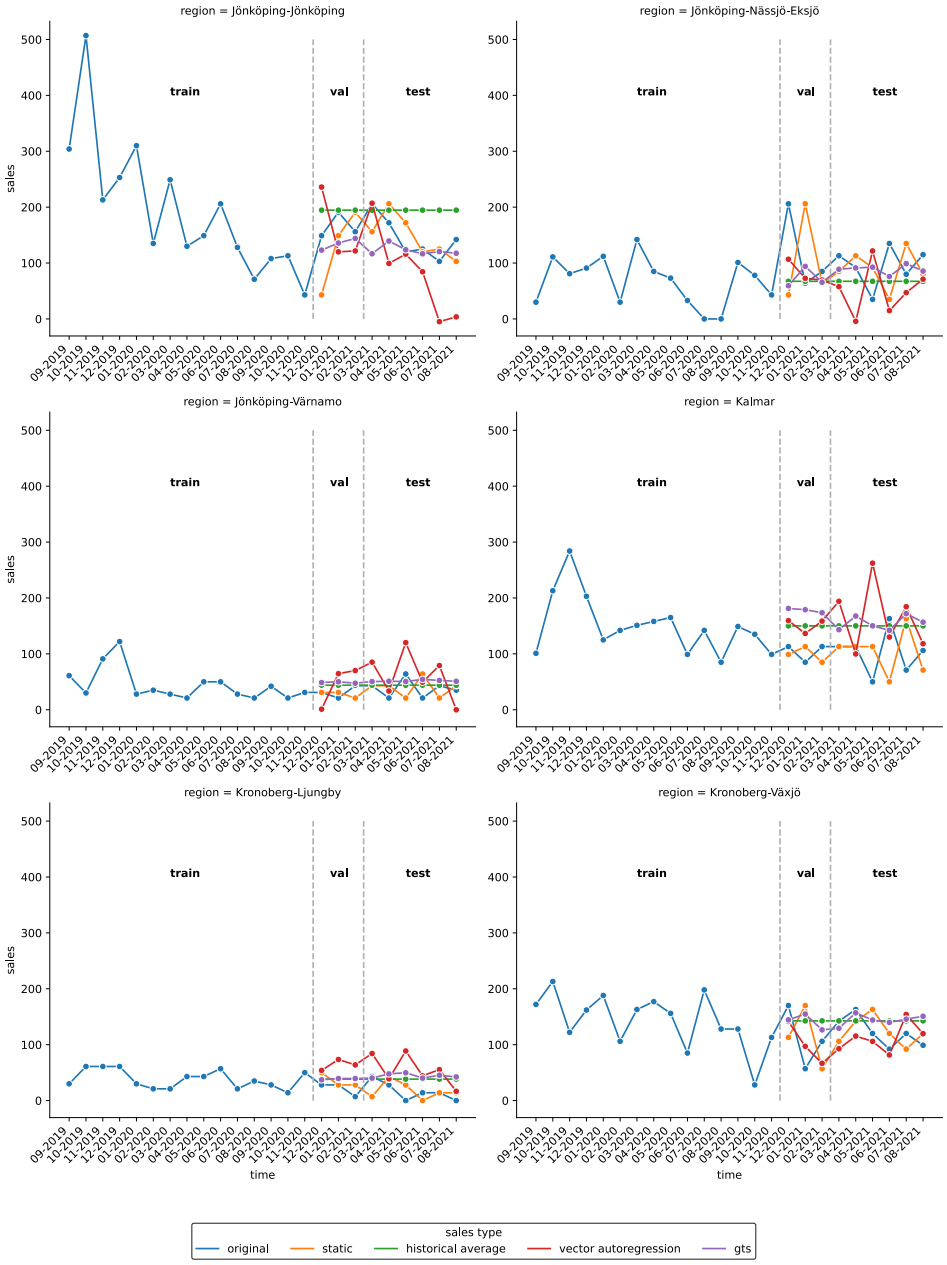


Figure 43: Sales forecasts for BC regions 7-12

Figure 44 plots the original Kisqali sales from 09-2019 to 08-2021 and the Kisqali sales forecasts from 12-2020 to 08-2021 for BC regions: Norrbotten-Sunderbyn, Skåne-Helsingborg-Landskrona, Skåne-Kristianstad, Skåne-Lund, Stockholm, Stockholm-Gotland.

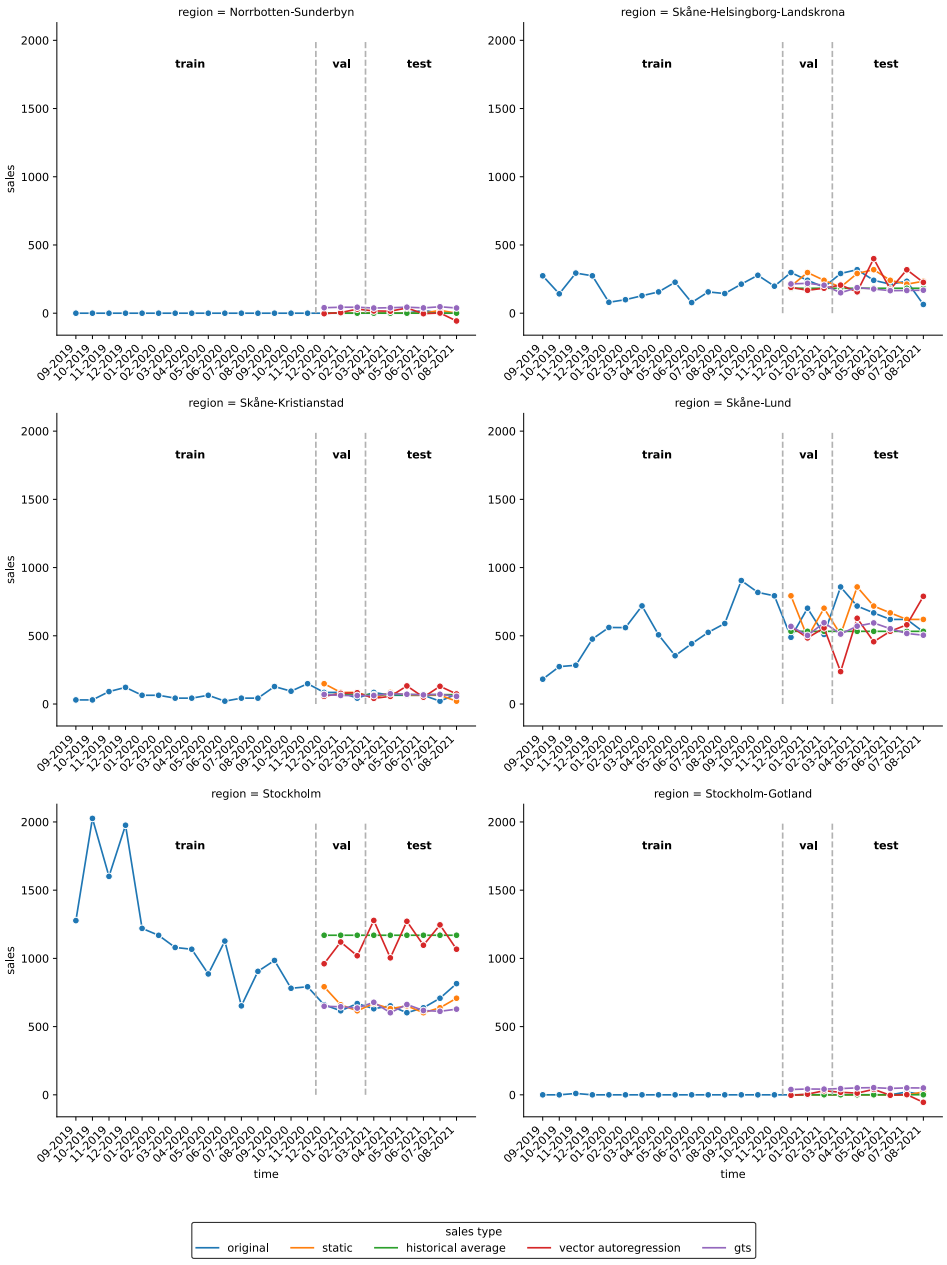


Figure 44: Sales forecasts for BC regions 13-18

Figure 45 plots the original Kisqali sales from 09-2019 to 08-2021 and the Kisqali sales forecasts from 12-2020 to 08-2021 for BC regions: Sörmland-Eskilstuna, Uppsala, Värmland-Karlstad, Västerbotten-Skellefteå, Västerbotten-Umeå, Västernorrland-Sundsvall.

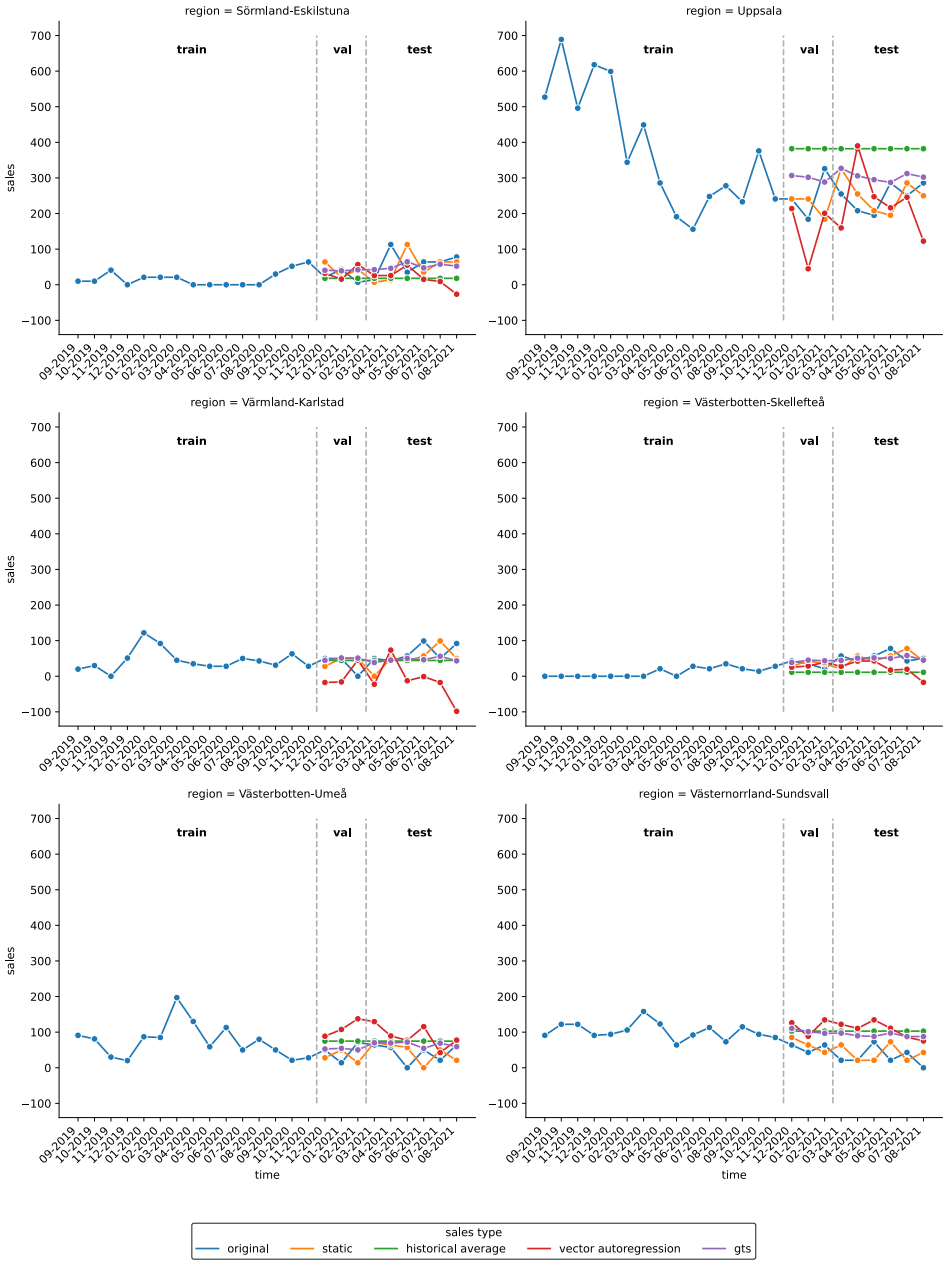


Figure 45: Sales forecasts for BC regions 19-24

Figure 46 plots the original Kisqali sales from 09-2019 to 08-2021 and the Kisqali sales forecasts from 12-2020 to 08-2021 for BC regions: Västernorrland-Örnsköldsvik, Västmanland-Västerås, Västra Götaland-Alingsås, Västra Götaland-Borås, Västra Götaland-Göteborg, Västra Götaland-Lidköping.

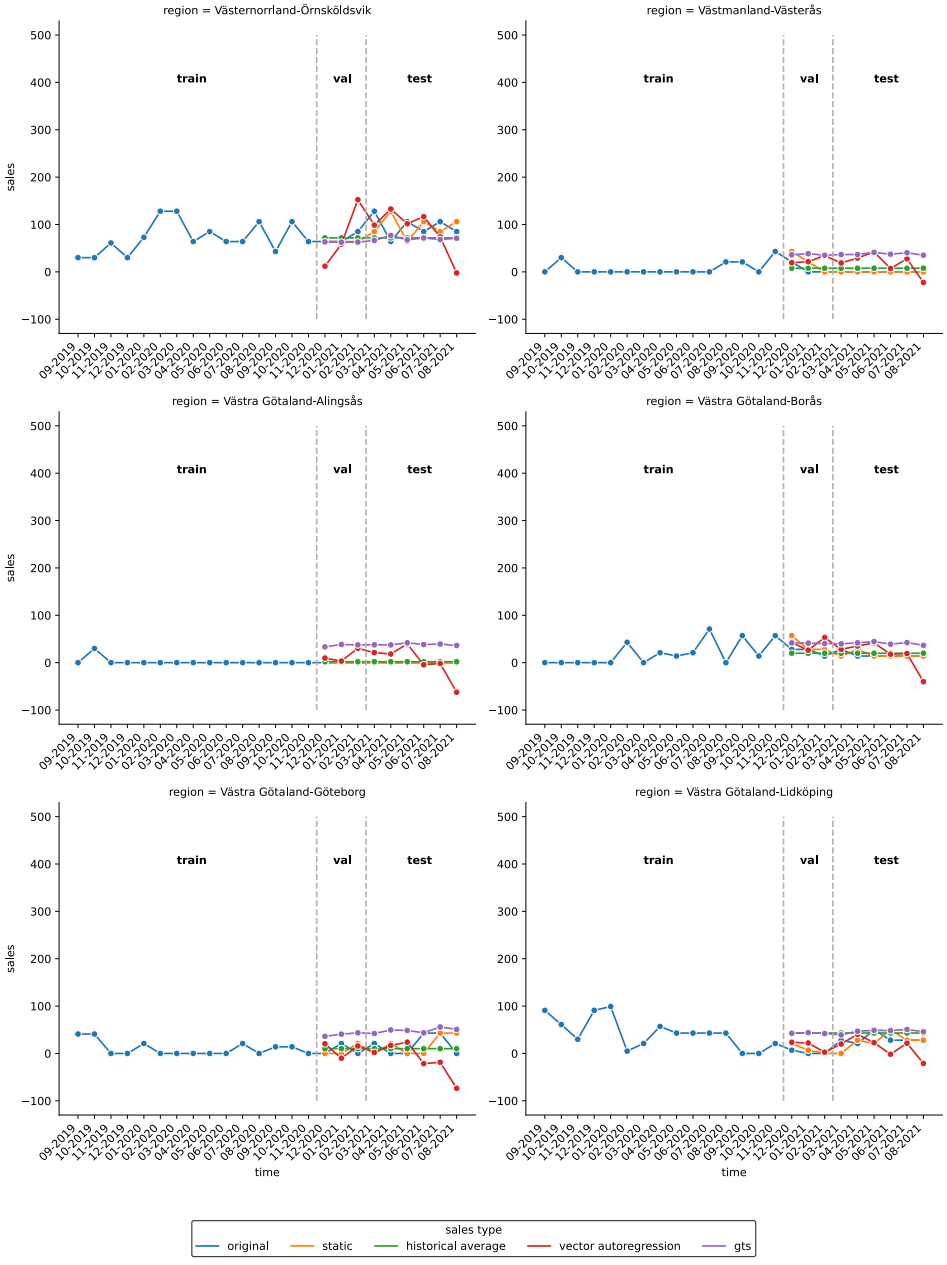


Figure 46: Sales forecasts for BC regions 25-30

Figure 47 plots the original Kisqali sales from 09-2019 to 08-2021 and the Kisqali sales forecasts from 12-2020 to 08-2021 for BC regions: Västra Götaland-Skövde, Västra Götaland-Uddevalla, Örebro-Örebro, Östergötland-Linköping, Östergötland-Norrköping.

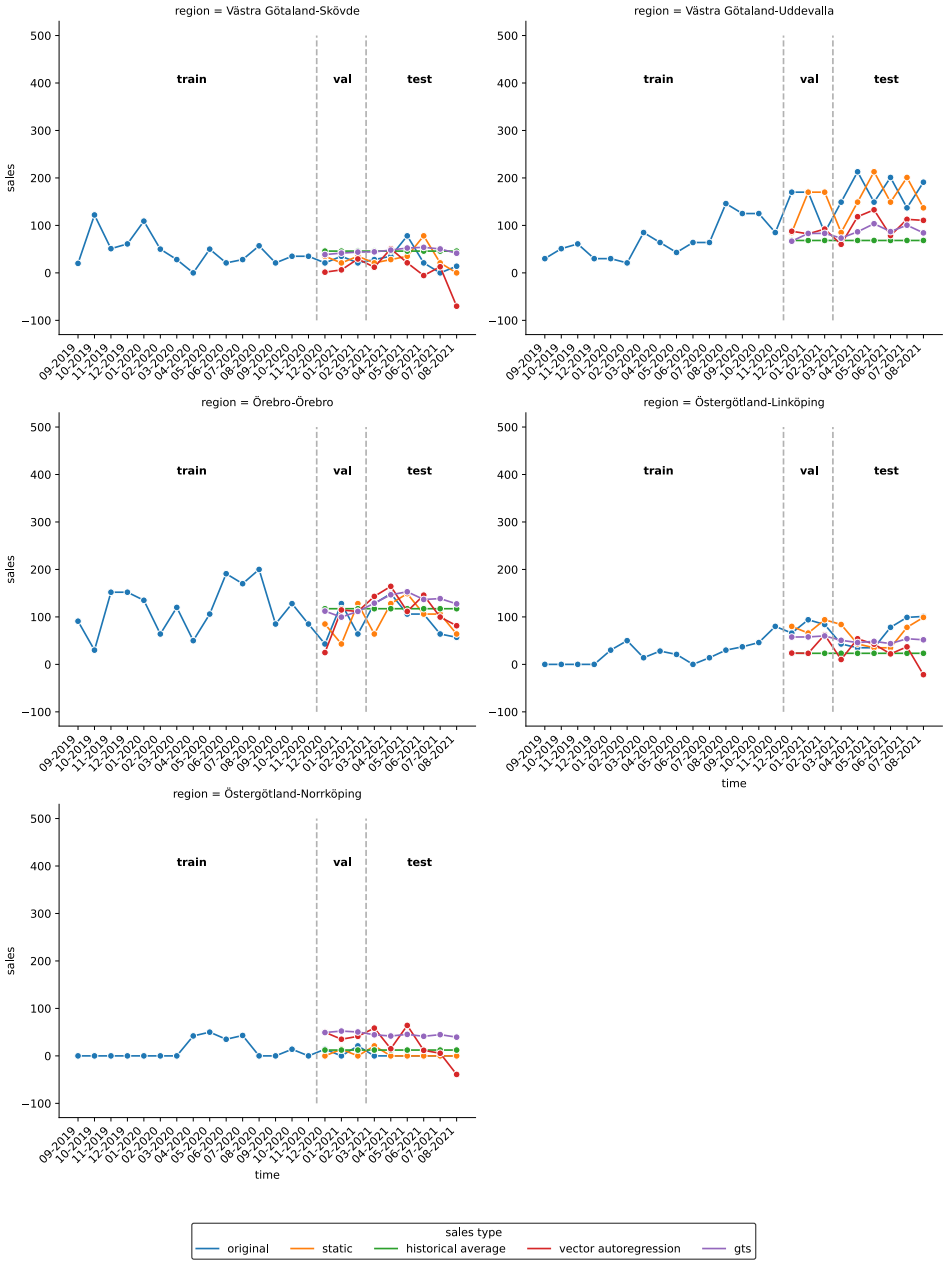


Figure 47: Sales forecasts for BC regions 31-35

Melanoma

The following four figures show the original Tafinlar-Mekinist sales from 09-2019 to 08-2021 in blue and the Tafinlar-Mekinist sales forecasts for all 22 BC regions. The static forecasts are orange, the historical average forecasts are green, the vector autoregression forecasts are red and the GTS forecasts are purple.

We have not generated forecasts for the sales in the training set, therefore there are no forecasts before 12-2020. The forecasts from 12-2020 to 02-2021 belong to the validation set and the forecasts from 03-2021 to 08-2021 to the test set.

Recall that the melanoma regions are those regions that bear the suffix "ONCO".

Figure 48 plots the original Tafinlar-Mekinist sales from 09-2019 to 08-2021 and the Tafinlar-Mekinist sales forecasts from 12-2020 to 08-2021 for melanoma regions: Blekinge ONCO, Dalarna ONCO, Gävleborg-Gävle ONCO, Halland-Halmstad ONCO, Jämtland ONCO, Jönköping ONCO.

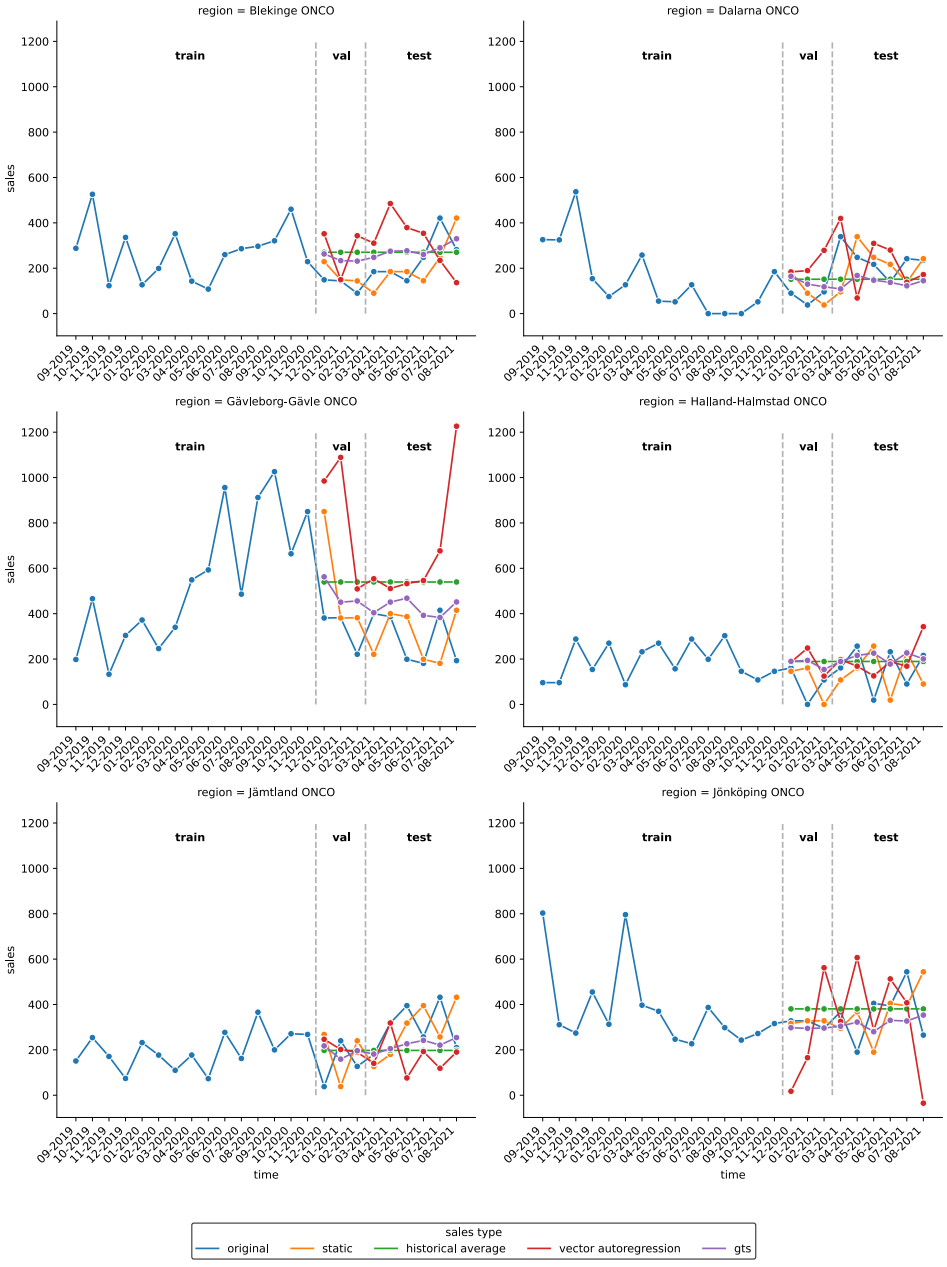


Figure 48: Sales forecasts for melanoma regions 1-6

Figure 49 plots the original Tafinlar-Mekinist sales from 09-2019 to 08-2021 and the Tafinlar-Mekinist sales forecasts from 12-2020 to 08-2021 for melanoma regions: Kalmar ONCO, Kronoberg-Växjö ONCO, Norrbotten-Sunderbyn ONCO, Skåne ONCO, Stockholm ONCO, Stockholm-Gotland ONCO.

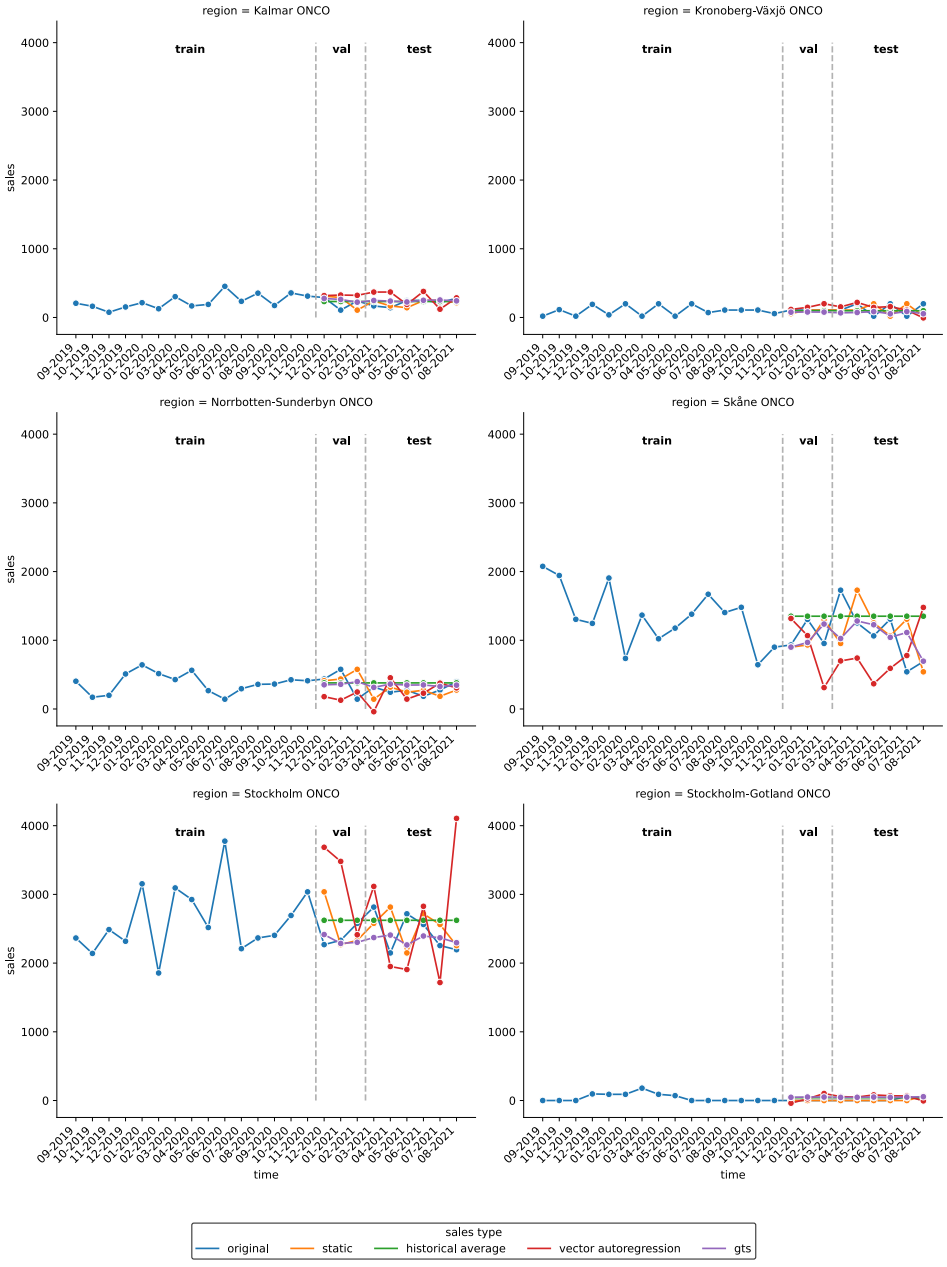


Figure 49: Sales forecasts for melanoma regions 7-12

Figure 50 plots the original Tafinlar-Mekinist sales from 09-2019 to 08-2021 and the Tafinlar-Mekinist sales forecasts from 12-2020 to 08-2021 for melanoma regions: Sörmland-Eskilstuna ONCO, Uppsala ONCO, Värmland-Karlstad ONCO, Västerbotten-Umeå ONCO, Västernorrland-Sundsvall ONCO, Västmanland-Västerås ONCO.

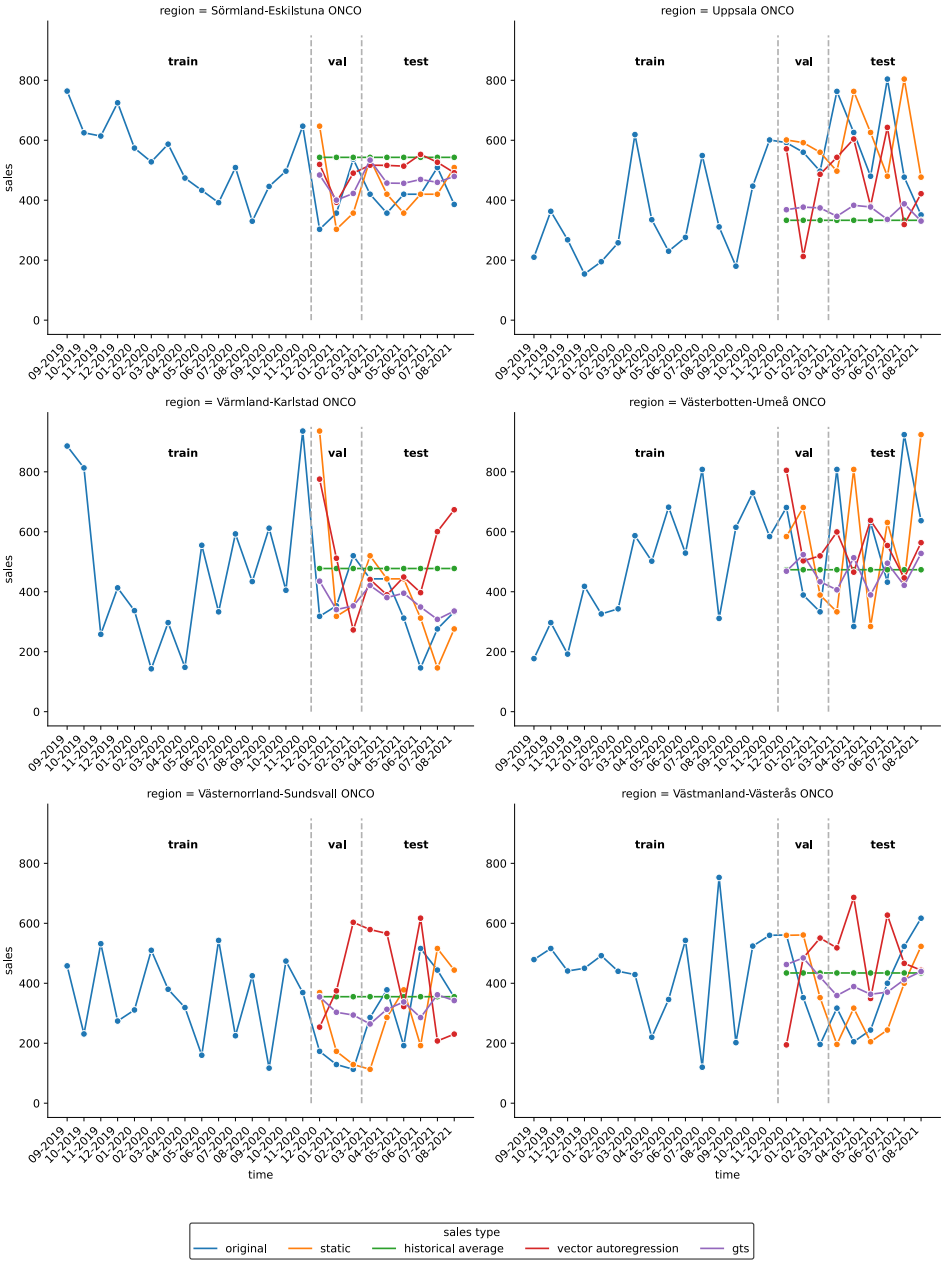


Figure 50: Sales forecasts for melanoma regions 13-18

Figure 51 plots the original Tafinlar-Mekinist sales from 09-2019 to 08-2021 and the Tafinlar-Mekinist sales forecasts from 12-2020 to 08-2021 for melanoma regions: Västra Götaland-Göteborg ONCO, Västra Götaland-SÅS ONCO, Örebro-Örebro ONCO, Östergötland-Linköping ONCO.

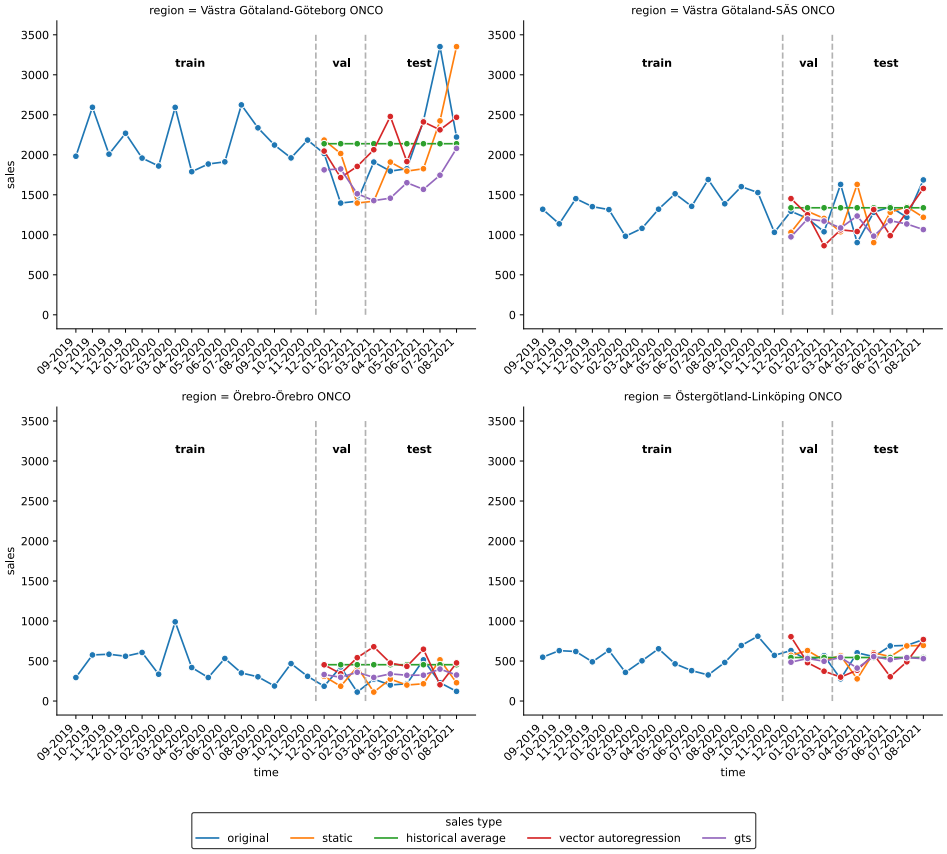


Figure 51: Sales forecasts for melanoma regions 19-22

Losses

Figure 52 shows for the three best hyperparameter combinations from Table 73 the training, validation and test losses.

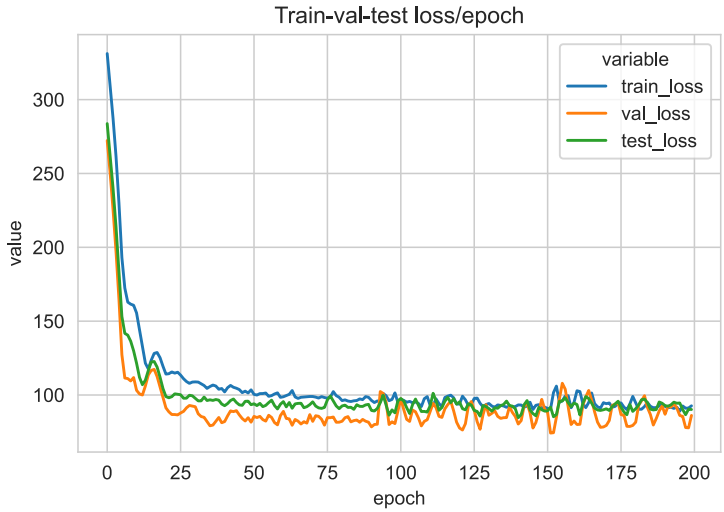


Figure 52: GTS: losses for three best hyperparameter combinations

Metrics

Figure 53 shows for the three best hyperparameter combinations from Table 73 the training, validation and test MAEs.

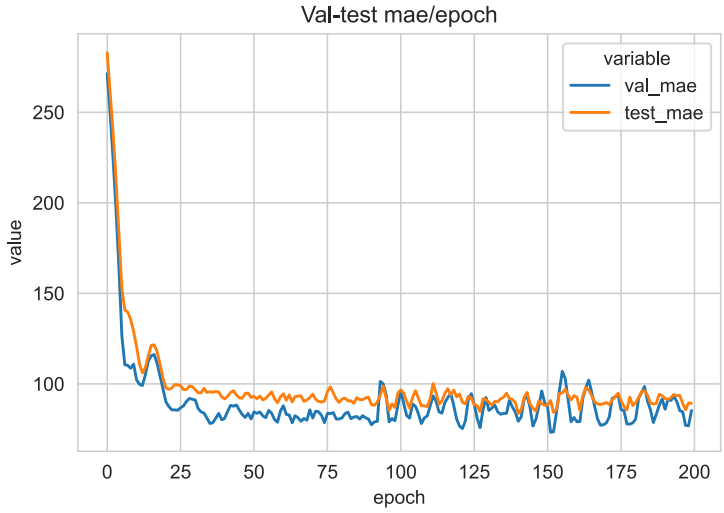


Figure 53: GTS: mean absolute errors for three best hyperparameter combinations

Figure 54 shows for the three best hyperparameter combinations from Table 73 the training, validation and test MAPEs.

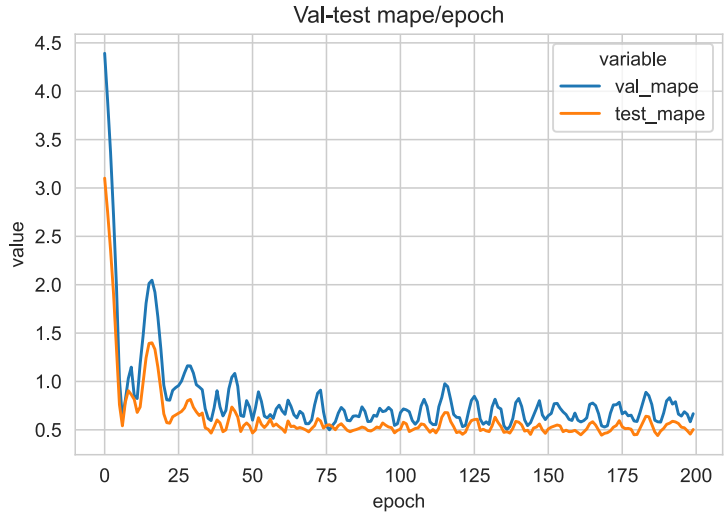


Figure 54: GTS: mean absolute percentage errors for three best hyperparameter combinations

Figure 55 shows for the three best hyperparameter combinations from Table 73 the training, validation and test RMSEs.

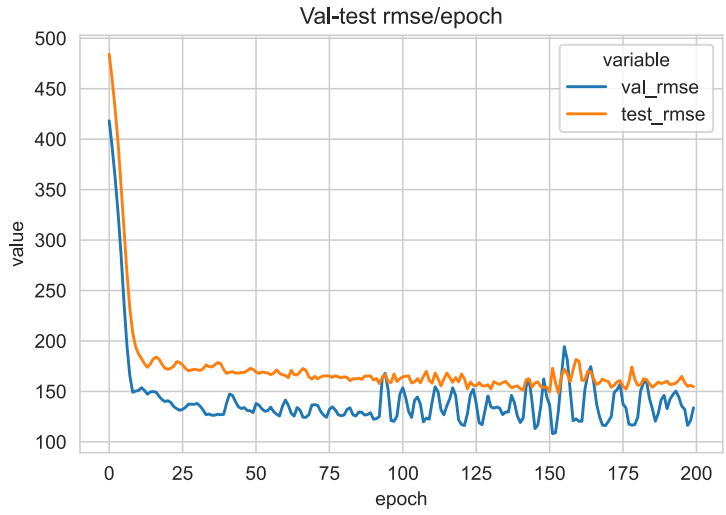


Figure 55: GTS: root mean square errors for three best hyperparameter combinations

A priori graph

In GTS we can incorporate an a priori graph into the training process. Figure 56 displays this a priori graph, which is given here by a simple kNN neighbors graph.

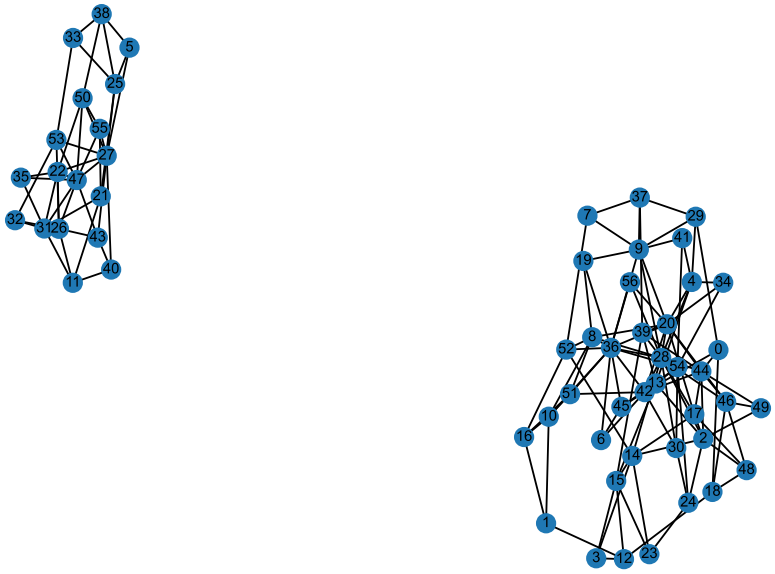


Figure 56: GTS: kNN neighbors graph

Table 82 summarizes some important properties of the a priori graph.

Graph property	Value
Nodes	57
Edges	151
Directed?	No
Weighted?	No
Isolated nodes	0
Self-loops	0
Density	0.0946
Clustering coefficient	0.3859
Min degree	3
Max degree	14
Avg degree	5.2982
Degree assortativity	0.0690
Number of connected components	2
Size of largest component	39 (68.42%)

Table 82: Summary of the kNN neighbors graph

Learned graph

During the training process, GTS learns a graph indicating how the 57 BC and melanoma regions are connected. It is intended to enhance the forecasts. Figure 57 shows this graph.

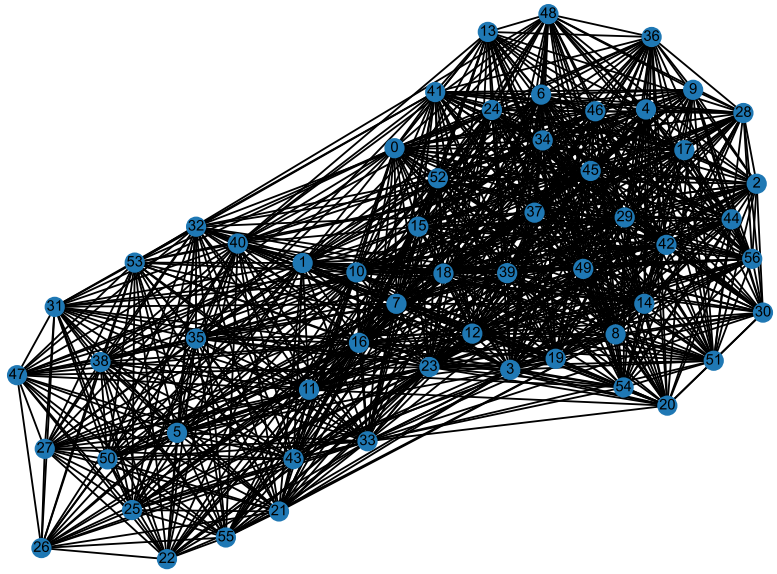


Figure 57: GTS: learned graph

Table 83 provides a summary of some important properties of the learned graph.

Graph property	Value
Nodes	57
Edges	925
Directed?	No
Weighted?	No
Isolated nodes	0
Self-loops	0
Density	0.5796
Clustering coefficient	0.7458
Min degree	20
Max degree	51
Avg degree	32.4561
Degree assortativity	0.0535
Number of connected components	1
Size of largest component	57 (100.00%)

Table 83: Summary of the learned graph

Bibliography

[1] Adamic, L. A. & Adar, E. (2003): *Friends and Neighbors on the Web.* Social Networks 25/3, pp. 211-230.

[2] Agrawal, M., Zitnik, M. & Leskovec, J. (2018): *Large-Scale Analysis of Disease Pathways in the Human Interactome.* Pacific Symposium on Biocomputing 2018, 3-7 January 2018, pp. 111-122.

[3] Ahmed, A., Shervashidze, N., Narayanamurthy, S., Josifovski, V. & Smola, A. J. (2013): *Distributed Large-Scale Natural Graph Factorization.* WWW 2013: Proceedings of the 22nd International Conference on World Wide Web, 13-17 May 2013, pp. 37-48.

[4] Al Hasan, M. & Zaki, M. J. (2011): *A Survey of Link Prediction in Social Networks.* In: Social Network Data Analytics, Springer, pp. 243-275.

[5] Asteriou, D. & Hall, S. G. (2011): *ARIMA Models and the Box-Jenkins Methodology.* In: Applied Econometrics, pp. 265-286. Palgrave MacMillan, London.

[6] Bahdanau, D., Cho, K. & Bengio, Y. (2015): *Neural Machine Translation by Jointly Learning to Align and Translate.* ICLR 2015: 3rd International Conference on Learning Representations, 7-9 May 2015.

[7] Barabási, A. L. & Albert, R. (1999): *Emergence of Scaling in Random Networks.* Science 286/5439, pp. 509-512.

[8] Baum, L. E. & Petrie, T. (1966): *Statistical Inference for Probabilistic Functions of Finite State Markov Chains.* The Annals of Mathematical Statistics 37/6, pp. 1554-1563.

[9] Belkin, M. & Niyogi, P. (2001): *Laplacian Eigenmaps and Spectral Techniques for Embedding and Clustering.* NIPS 2001: Proceedings of the 14th International Conference on Neural Information Processing Systems, 3-8 December 2001, pp. 585-591.

[10] Bennett, J. & Lanning, S. (2007): *The Netflix Prize.* Proceedings of the KDD Cup Workshop 2007, pp. 3-6.

[11] Bento, C. (2021): *Multilayer Perceptron Explained with a Real-Life Example and Python Code: Sentiment Analysis.* Available at: `https://towardsdatascience.com/`

multilayer-perceptron-explained-with-a-real-life-example-and-python-code-sentiment-analysis-cb408ee93141, last accessed on 21 July 2022.

[12] Bordes, A., Usunier, N., Garcia-Duran, A., Weston, J. & Yakhnenko, O. (2013): *Translating Embeddings for Modeling Multi-Relational Data.* NIPS 2013: Advances in Neural Information Processing Systems 26.

[13] Brin, S. & Page, L. (2012): Reprint of: *The Anatomy of a Large-Scale Hypertextual Web Search Engine.* Computer Networks 56/18, pp. 3825-3833.

[14] Cancino, F. (2021): *What is Logistic Regression?.* Available at: https://towardsdatascience.com/what-is-logistic-regression-d3f59ef1a9b, last accessed on 21 July 2022.

[15] Cangea, C., Veličković, P., Jovanović, N., Kipf, T. & Liò, P. (2013): *Towards Sparse Hierarchical Graph Classifiers.* Version 1. 3 November 2018. arXiv: 1811.01287.

[16] Cao, D., Wang, Y., Duan, J., Zhang, C. Zhu, X., Huang, C., Tong, Y., Xu, B., Bai, J., Tong, J. & Zhang, Q. (2020): *Spectral Temporal Graph Neural Network for Multivariate Time-series Forecasting.* NeurIPS 2020: 34th Conference on Neural Information Processing Systems, 6-12 December 2020.

[17] Cho, K., van Merrienboera, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H. & Bengio, Y. (2014): *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation.* EMNLP 2014: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, 25-29 October 2014, pp. 1724–1734.

[18] Cohen, E. (2018): *node2vec: Embeddings for Graph Data.* Available at: https://towardsdatascience.com/node2vec-embeddings-for-graph-data-32a866340fef, last accessed on 22 July 2022.

[19] Currie Sivek, S. (2021): *Notable Nodes: Identifying Influencers with Network Analysis.* Available at: https://towardsdatascience.com/notable-nodes-identifying-influencers-with-network-analysis-2f51f1d8fec4, last accessed on 21 July 2022.

[20] Defferrard, M., Bresson, X. & Vandergheynst, P. (2016): *Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering.* NIPS 2016: Advances in Neural Information Processing Systems 29, 5-10 December 2016.

[21] Devlin, J., Chang, M.-W., Lee, K. & Toutanova, K. (2019): *BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding.* NAACL-HLT 2019: Proceedings of the 17th Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, 2-7 June 2019, pp. 4171-4186.

[22] Disney, A. (2020): *Social network analysis 101: centrality measures explained.* Available at: `https://cambridge-intelligence.com/keylines-faqs-social-network-analysis/`, last accessed on 21 July 2022.

[23] Dissanayake, B. (2020): *Vector Auto Regression for Multivariate Time Series Forecasting.* Available at: `https://towardsdatascience.com/vector-auto-regression-for-multivariate-time-series-forecasting-cf5216c9ce23`, last accessed on 23 July 2022.

[24] Donnat, C., Zitnik, M., Hallac, D. & Leskovec, J. (2018): *Learning Structural Node Embeddings via Diffusion Wavelets.* KDD 2018: Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, 19-23 August 2018, pp. 1320-1329.

[25] Elinas, P. (2019): *Knowing Your Neighbors: Machine Learning on Graphs.* Available at: `https://medium.com/stellargraph/knowing-your-neighbours-machine-learning-on-graphs-9b7c3d0d5896`, last accessed on 14 July 2022.

[26] Fey, M. & Lenssen, J. E. (2019): *Fast Graph Representation Learning with PyTorch Geometric.* Workshop paper, 7th International Conference on Learning Representations, 6-9 May 2019.

[27] Franceschi, L., Donini, M., Frasconi, P. & Pontil, M. (2017): *Forward and Reverse Gradient-Based Hyperparameter Optimization.* ICML 2017: Proceedings of the 34th International Conference on Machine Learning, 6-11 August 2017, pp. 1165–1173.

[28] Franceschi, L., Niepert, M., Pontil, M. & He, X. (2019): *Learning Discrete Structures for Graph Neural Networks.* ICML 2019: Proceedings of the 36th International Conference on Machine Learning, 10-15 June 2019, pp. 1972-1982.

[29] Gandhi, R. (2018): *Support Vector Machine - Introduction to Machine Learning Algorithms.* Available at: `https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47`, last accessed on 21 July 2022.

[30] Gilmer, J., Schoenholz, S., Riley, P., Vinyals, O. & Dahl, G. (2017): *Neural Message Passing for Quantum Chemistry*. ICML 2017: Proceedings of the 34th International Conference on Machine Learning, 6-11 August 2017, pp. 1263-1272.

[31] Granger, C. W. J. (1969): *Investigating Causal Relations by Econometric Models and Cross-Spectral Methods*. Econometrica 37/3, pp. 424-438.

[32] Hagberg, A. A., Schult, D. A. & Swart, P. J. (2008): *Exploring network structure, dynamics, and function using NetworkX*. SciPy 2008: Proceedings of the 7th Python in Science Conference, 19-24 August 2008, pp. 11–15.

[33] Hamilton, J. D. (2014): *Time Series Analysis*. Princeton University Press, Princeton.

[34] Hamilton, W. L. (2020): *Graph Representation Learning*. Morgan & Claypool Publishers, San Rafael.

[35] Hamilton, W. L., Ying, R. & Leskovec, J. (2017): *Inductive Representation Learning on Large Graphs*. NeurIPS 2017: 31st Conference on Neural Information Processing Systems, 4-9 December 2017.

[36] Hammond, D. K., Vandergheynst, P. & Gribonval, R. (2011): *Wavelets on Graphs via Spectral Graph Theory*. Applied and Computational Harmonic Analysis 30/2, pp. 129–150.

[37] Herman, M. [mfherman] (2020): *I haven't used ggmap much, but my understanding is that it just produces a ggplot object, so you should be* [Comment on the online forum post *ggmap, tidy verse: "non-USA countries" with county, municipality, subregion data?*]. RStudio Community. Available at: `https://community.rstudio.com/t/ggmap-tidy-verse-non-usa-countries-with-county-municipality-subregion-data/68527`, last accessed on 18 July 2022.

[38] Heydt, M. (2018): *Python Web Scraping Cookbook*. Packt Publishing, Birmingham.

[39] Hochreiter, S. & Schmidhuber, J. (1997): *Long Short-Term Memory*. Neural Computation 9/8, pp. 1735-1780.

[40] Hu, L. (2018): *link-prediction*. GitHub repository. Available at: `https://github.com/lucashu1/link-prediction`.

[41] Jang, E., Gu, S. & Poole, B. (2017): *Categorical Reparameterization with Gumbel-Softmax*. ICLR 2017: 5th International Conference on Learning Representations, 24-26 April 2017.

[42] Jeh, G. & Widom, J. (2002): *SimRank: A Measure of Structural-Context Similarity*. KDD 2002: Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 23-26 July 2002, pp. 538-543.

[43] Katz, L. (1953): *A New Status Index Derived from Sociometric Analysis*. Psychometrika 18/1, pp. 39–43.

[44] Kingma, D. P. & Welling, M. (2014): *Auto-Encoding Variational Bayes*. ICLR 2014: 2nd International Conference on Learning Representations, 14-16 April 2014.

[45] Kipf, T. N., Fetaya, E., Wang, K.-C., Welling, M. & Zemel, R. (2018): *Neural Relational Inference for Interacting Systems*. ICLR 2018: Proceedings of the 35th International Conference on Machine Learning, 30 April - 3 May 2018.

[46] Kipf, T. N. & Welling, M. (2016): *Variational Graph Auto-Encoders*. NIPS 2016: Bayesian Deep Learning Workshop, 10 December 2016.

[47] Kipf, T. N. & Welling, M. (2017): *Semi-Supervised Classification with Graph Convolutional Networks*. Conference paper, 5th International Conference on Learning Representations, 24-26 April 2017.

[48] Koren, Y. (2008): *Factorization Meets the Neighborhood: a Multifaceted Collaborative Filtering Model*. KDD 2008: Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 24-27 August 2008, pp. 426–434.

[49] Koren, Y., Bell, R. & Volinsky, C. (2009): Matrix Factorization Techniques for Recommender Systems. Computer 42/8, pp. 30-37.

[50] Kron, N. (2020): *svenska-stader*. GitHub repository. Available at: `https://github.com/sphrak/svenska-stader`.

[51] Lachapelle, S., Brouillard, P., Deleu, T. & Lacoste-Julien, S. (2020): *Gradient-Based Neural DAG Learning*. ICLR 2020: 8th International Conference on Learning Representations, 26 April - 1 May 2020.

[52] Li, Q., Han, Z. & Wu, X.-M. (2018a): *Deeper Insights into Graph Convolutional Networks for Semi-Supervised Learning*. AAAI 2018: Proceedings of the 32nd AAAI Conference on Artificial Intelligence, 2-7 February 2018, pp. 3538–3545.

[53] Li, Y., Yu, R., Shahabi, C. & Liu, Y. (2018b): *Diffusion Convolutional Recurrent Neural Network: Data-Driven Traffic Forecasting*. ICLR 2018: 6th International Conference on Learning Representations, 30 April - 3 May 2018.

[54] Li, G., Müller, M., Thabet, A. & Ghanem, B. (2019a): *DeepGCNs: Can GCNs Go as Deep as CNNs?*. ICCV 2019: Proceedings of the IEEE/CVF International Conference on Computer Vision, 27 October - 2 November 2019.

[55] Li, S., Jin, X., Xuan, Y., Zhou, X., Chen, W., Wang, Y.-X. & Yan, X. (2019b): *Enhancing the Locality and Breaking the Memory Bottleneck of Transformer on Time Series Forecasting*. NeurIPS 2019: Proceedings of the 33rd International Conference on Neural Information Processing Systems, 8-14 December 2019, pp. 5243–5253.

[56] Li, Y., Gu, C., Dullien, T., Vinyals, O. & Kohli, P. (2019c): *Graph Matching Networks for Learning the Similarity of Graph Structured Objects*. ICML 2019: Proceedings of the 36th International Conference on Machine Learning, 10-15 June 2019, pp. 3835-3845.

[57] Liao, R., Zhao, Z., Urtasun, R. & Zemel, R. S. (2019): *LanczosNet: Multi-Scale Deep Graph Convolutional Networks*. ICLR 2019: 7th International Conference on Learning Representations, 6-9 May 2019.

[58] Liben-Nowell, D. & Kleinberg, J. (2007): *The Link Prediction Problem for Social Networks*. Journal of the American Society for Information Science and Technology 58/7, pp. 1019–1031.

[59] Liu, Z. & Zhou, J. (2020): *Introduction to Graph Neural Networks*. Morgan & Claypool Publishers, San Rafael.

[60] Lops, P., De Gemmis, M. & Semeraro, G. (2011): *Content-Based Recommender Systems: State of the Art and Trends*. In: Recommender Systems Handbook, Springer, pp. 73-105.

[61] Lü, L. & Zhou, T. (2011): *Link Prediction in Complex Networks: A Survey*. Physica A: Statistical Mechanics and its Applications 390/6, pp. 1150-1170.

[62] Maddison, C. J., Mnih, A. & Teh, Y. W. (2017): *The Concrete Distribution: A Continuous Relaxation of Discrete Random Variables*. ICLR 2017: 5th International Conference on Learning Representations, 24-26 April 2017.

[63] Makridakis, S. G., Wheelwright, S. C. & Hyndman, R. J. (1997): *Forecasting: Methods and Applications*. 3rd edition. Wiley India Pvt. Ltd, New Delhi.

[64] McPherson, M., Smith-Lovin, L. & Cook, J. (2001): *Birds of a Feather: Homophily in Social Networks*. Annual Review of Sociology 27/1, pp. 415-444.

[65] Mikolov, T., Chen, K., Corrado, G. & Dean, J. (2013): *Efficient Estimation of Word Representations in Vector Space*. Version 3. 7 September 2013. arXiv 1301.3781.

[66] Moreno, M. & Basille, M. (2018): *Drawing beautiful maps programmatically with R, sf and ggplot2 - Part 2: Layers*. Available at: https://r-spatial.org/r/2018/10/25/ggplot2-sf-2.html, last accessed on 18 July 2022.

[67] Nickel, M., Jiang, X. & Tresp, V. (2014): *Reducing the Rank in Relational Factorization Models by Including Observable Patterns*. NIPS 2014: Advances in Neural Information Processing Systems 27, 8-13 December 2014, pp. 1179-1187.

[68] Nickel, M., Murphy, K., Tresp, V. & Gabrilovich, E. (2016): *A Review of Relational Machine Learning for Knowledge Graphs*. Proceedings of the IEEE 104/1, pp. 11-33.

[69] Ou, M., Cui, P., Pei, J., Zhang, Z. & Zhu, W. (2016): *Asymmetric Transitivity Preserving Graph Embedding*. KDD 2016: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 13-17 August 2016, pp. 1105–1114.

[70] Oyetunde, T., Zhang, M., Chen, Y., Tang, Y. J. & Lo, C. (2017): *BoostGAPFILL: Improving the Fidelity of Metabolic Network Reconstructions through Integrated Constraint and Pattern-Based Methods*. Bioinformatics 33/4, pp. 608–611.

[71] Pandit, S., Chau, D., Wang, S. & Faloutsos, C. (2007): *NetProbe: A Fast and Scalable System for Fraud Detection in Online Auction Networks*. WWW 2007: Proceedings of the 16th International Conference on World Wide Web, 8-12 May 2007, pp. 201-210.

[72] Pant, A. (2019): *Introduction to Logistic Regression*. Available at: https://towardsdatascience.com/introduction-to-logistic-regression-66248243c148, last accessed on 21 July 2022.

[73] Paszke, A., Gross, S. Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J. & Chintala, S. (2019): *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. In: Advances in Neural Information Processing Systems 32. Curran Associates, Inc., pp. 8024–8035. Available at: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

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

[75] Perozzi, B., Al-Rfou, R. & Skiena, S. (2014): *Online Learning of Social Representations.* KDD 2014: Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 24-27 August 2014, pp. 701–710.

[76] Qi, Y., Bar-Joseph, Z. & Klein-Seetharaman, J. (2006): *Evaluation of Different Biological Data and Computational Classification Methods for Use in Protein Interaction Prediction.* Proteins: Structure, Function, and Bioinformatics 63/3, pp. 490–500.

[77] Qiu, J., Dong, Y., Ma, H., Li, J., Wang, K. & Tang, J. (2018): *Network Embedding as Matrix Factorization: Unifying DeepWalk, LINE, PTE, and node2vec.* WSDM 2018: Proceedings of the 11th ACM International Conference on Web Search and Data Mining, 5-9 February 2018, pp. 459–467.

[78] Řehůřek, R. & Sojka, P. (2010): *Software Framework for Topic Modelling with Large Corpora.* Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks, pp. 45-50.

[79] Reitz, K. (2022): *Requests Documentation.* Available at: `https://buildmedia.readthedocs.org/media/pdf/requests/latest/requests.pdf`, last accessed on 20 July 2022.

[80] Rendle, S. (2010): *Factorization Machines.* ICDM 2010: 10th IEEE International Conference on Data Mining, 13-17 December 2010, pp. 995–1000.

[81] Ribeiro, L. F., Saverese, P. H. & Figueiredo, D. R. (2017): *struc2vec: Learning Node Representations from Structural Identity.* KDD 2017: Proceedings of the 23th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 13-17 August 2017, pp. 385–394.

[82] Richardson, L. (2020): *Beautiful Soup Documentation.* Available at: `https://www.crummy.com/software/BeautifulSoup/bs4/doc/`, last accessed on 20 July 2022.

[83] Rong, Y., Huang, W., Xu, T. & Huang, J. (2020): *DropEdge: Towards Deep Graph Convolutional Networks on Node Classification.* ICLR 2020: 8th International Conference on Learning Representations, 26 April - 1 May 2020.

[84] Sanders, R. (2022): *The 5 Types of Influencers You Need to Know.* Available at: `https://www.simplilearn.com/types-of-influencers-article`, last accessed on 21 July 22.

[85] Schuster, M. & Paliwal, K. K. (1997): *Bidirectional Recurrent Neural Networks.* IEEE Transactions on Signal Processing 45/11, pp. 2673-2681.

[86] Seabold, S. & Perktold, J. (2010): *Statsmodels: Econometric and statistical modeling with Python.* Proceedings of the 9th Python in Science Conference.

[87] Seo, Y., Defferrard, M., Vandergheynst, P. & Bresson, X. (2016): *Structured Sequence Modeling with Graph Convolutional Recurrent Networks.* Version 1. 22 December 2016. arXiv: `1612.07659`.

[88] Shang, C., Chen, J. & Bi, J. (2021): *Discrete Graph Structure Learning for Forecasting Multiple Time Series.* Conference paper, 9th International Conference on Learning Representations, 3-7 May 2021.

[89] Shibata, N., Kajikawa, Y. & Sakata, I. (2012): *Link Prediction in Citation Networks.* Journal of the American Society for Information Science and Technology 63/1, pp. 78–85.

[90] Shih, S.-Y., Sun, F.-K. & Lee, H.-y. (2019): *Temporal Pattern Attention for Multivariate Time Series Forecasting.* Machine Learning 108/8-9, pp. 1421-1441.

[91] Socialstyrelsen (2021a): *Statistical Database, Cancer.* [Statistikdatabasen_Breast Cancer_17_09_2021 14_15_07.xlsx]. Available at: `https://sdb.socialstyrelsen.se/if_can/val_eng.aspx`.

[92] Socialstyrelsen (2021b): *Statistical Database, Cancer.* [Statistikdatabasen_Malignant Melanoma 17_09_2021 14_13_18.xlsx]. Available at: `https://sdb.socialstyrelsen.se/if_can/val_eng.aspx`.

[93] Stanfield, Z., Coşkun, M. & Koyutürk, M. (2017): *Drug Response Prediction as a Link Prediction Problem.* Scientific Reports 7/1, pp. 1-13.

[94] Statistics Sweden (2021): *Population 1 November by region, age and sex. Year 2002 - 2021.* [Population by Age 2020.xlsx]. Available at: `https://www.statistikdatabasen.scb.se/pxweb/en/ssd/START__BE__BE0101__BE0101A/FolkmangdNov/`.

[95] Staudt, C., Sazonovs, A. & Meyerhenke, H. (2016): *NetworKit: A Tool Suite for Large-scale Complex Network Analysis.* Network Science 4/4, pp. 508-530, Cambridge University Press.

[96] Sutskever, I., Vinyals, O. & Le, Q. V. (2014): *Sequence to Sequence Learning with Neural Networks.* NIPS 2014: Proceedings of the 27th International Conference on Neural Information Processing Systems, 8-13 December 2014, pp. 3104–3112.

[97] Tang, J., Qu, M., Wang, M., Zhang, M., Yan, J. & Mei, Q. (2015): *LINE: Large-Scale Information Network Embedding.* WWW 2015: Proceedings of the 24th International Conference on World Wide Web, 18-22 May 2015, pp. 1067–1077.

[98] Teru, K., Denis, E. & Hamilton, W. L. (2020): *Inductive Relation Prediction on Knowledge Graphs.* ICML 2020: 37th International Conference on Machine Learning, 12-18 July 2020.

[99] vanden Broucke, S. & Baesens, B. (2018): *Practical Web Scraping for Data Science.* Apress, New York.

[100] van der Maaten, L. & Hinton, G. (2008): *Visualizing Data using t-SNE.* Journal of Machine Learning Research 9, pp. 2579-2605.

[101] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł. & Polosukhin, I. (2017): *Attention is All You Need.* NIPS 2017: Advances in Neural Information Processing Systems 30, 4-9 December 2017.

[102] Veličković, P., Cucurull, G., Casanova, A., Romero, A., Liò, P. & Bengio, Y. (2018): *Graph Attention Networks.* ICLR 2018: 6th International Conference on Learning Representations, 30 April - 3 May 2018.

[103] Veličković, P., Fedus, W., Hamilton, W. L., Liò, P., Bengio, Y. & Hjelm, R. D. (2019): *Deep Graph Infomax.* ICLR 2019: 7th International Conference on Learning Representations, 6-9 May 2019.

[104] von Luxburg, U. (2007): *A Tutorial on Spectral Clustering.* Statistics and Computing 17/4, pp. 395-416.

[105] Waikhom, L. & Patgiri, R. (2021): *Graph Neural Networks: Methods, Applications, and Opportunities.* Version 2. 8 September 2021. arXiv: 2108.10733.

[106] Wang, Y., Zhao, Y., Shah, N. & Derr, T. (2021): *Imbalanced Graph Classification via Graph-of-Graph Neural Networks.* Version 1. 1 December 2021. arXiv: 2112.00238.

[107] Williams, R. J., Hinton, G. E. & Rumelhart, D. E. (1986): *Learning Representations by Back-Propagating Errors.* Nature 323/6088, pp. 533-536.

[108] Wobben, G. (2019): *pymed.* Available at https://pypi.org/project/pymed/, last accessed on 19 July 2022.

[109] Wu, L., Cui, P., Pei, J. & Zhao, L. (2022): *Graph Neural Networks: Foundations, Frontiers, and Applications.* Springer Singapore, Singapore.

[110] Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C. & Yu, P. S. (2021): *A Comprehensive Survey on Graph Neural Networks.* IEEE Transactions on Neural Networks and Learning Systems 32/1, pp. 4-24.

[111] Wu, Z., Pan, S., Long, G., Jiang, J., Chang, X. & Zhang, C. (2020): *Connecting the Dots: Multivariate Time Series Forecasting with Graph Neural Networks.* KDD 2020: Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 23-27 August 2020, pp. 753–763.

[112] Xhonneux, L.-P. A. C., Qu, M. & Tang, J. (2020): *Continuous Graph Neural Networks.* ICML 2020: Proceedings of the 37th International Conference on Machine Learning, 12-18 July 2020, pp. 10432-10441.

[113] Xu, K., Li, C., Tian, Y., Sonobe, T., Kawarabayashi, K. & Jegelka, S. (2018): *Representation Learning on Graphs with Jumping Knowledge Networks.* ICML 2018: 35th International Conference on Machine Learning, 10-15 July 2018, pp. 5453–5462.

[114] Xu, K., Hu, W., Leskovec, J. & Jegelka, S. (2019): *How Powerful are Graph Neural Networks?.* ICLR 2019: 7th International Conference on Learning Representations, 6-9 May 2019.

[115] Yang, Z., Cohen, W. W. & Salakhutdinov, R. (2016): *Revisiting Semi-Supervised Learning with Graph Embeddings.* ICML 2016: Proceedings of the 33rd International Conference on Machine Learning, 19-24 June 2016, pp. 40-48.

[116] Ying, R., He, R., Chen, K., Eksombatchai, P., Hamilton, W. L. & Leskovec, J. (2018): *Graph Convolutional Neural Networks for Web-Scale Recommender Systems.* KDD 2018: Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 19-23 August 2018, pp. 974-983.

[117] Yu, B., Yin, H. & Zhu, Z. (2018): *Spatio-Temporal Graph Convolutional Networks: A Deep Learning Framework for Traffic Forecasting.* IJCAI 2018: Proceedings of the 27th International Joint Conference on Artificial Intelligence, 13-19 July 2018.

[118] Yu, Y., Chen, J., Gao, T. & Yu, M. (2019): *DAG-GNN: DAG Structure Learning with Graph Neural Networks.* ICML 2019: Proceedings of the 36th International Conference on Machine Learning, 10-15 June 2019.

[119] Yuan, S., Wang, C., Jiang, Q. & Ma, J. (2022): *Community Detection with Graph Neural Network using Markov Stability.* Conference paper, 2022 International Conference on Artificial Intelligence in Information and Communication (ICAIIC), 21-24 February 2022, pp. 437-442.

[120] Zarchan, P. & Musoff, H. (2000): *Fundamentals of Kalman Filtering.* American Institute of Aeronautics and Astronautics, Incorporated.

[121] Zhang, M. & Chen, Y. (2018): *Link Prediction Based on Graph Neural Networks.* NeurIPS 2018: Proceedings of the 32nd International Conference on Neural Information Processing Systems, 2-8 December 2018, pp. 5171-5181.

[122] Zhang, M., Cui, Z., Neumann, M. & Chen, Y. (2018): *An End-to-End Deep Learning Architecture for Graph Classification.* AAAI 2018: Proceedings of the 32nd AAAI Conference on Artificial Intelligence, 2-7 February 2018, pp. 4438-4445.

[123] Zhao, L. & Akoglu, L. (2020): *PairNorm: Tackling Oversmoothing in GNNs.* ICLR 2020: 8th International Conference on Learning Representations, 26 April - 1 May 2020.

[124] Zhao, H., Du, L. & Buntine, W. (2017): *Leveraging Node Attributes for Incomplete Relational Data.* ICML 2017: Proceedings of the 34th International Conference on Machine Learning, 6-11 August 2017, pp. 4072-4081.

[125] Zhao, L., Song, Y., Zhang, C., Liu, Y., Wang, P., Lin, T., Deng, M. & Li, H. (2019): *T-GCN: A Temporal Graph Convolutional Network for Traffic Prediction.* IEEE Transactions on Intelligent Transportation Systems 21/9, pp. 3848-3858.

[126] Zheng, X., Aragam, B., Ravikumar, P. & Xing, E. P. (2018): *DAGs with NO TEARS: Continuous Optimization for Structure Learning.* NIPS 2018: Proceedings of the 32nd International Conference on Neural Information Processing Systems, 3-8 December 2018, pp. 9492–9503.

[127] Zhou, D., Bousquet, O., Lal, T. N., Weston, J. & Schölkopf, B. (2004): *Learning with Local and Global Consistency.* NIPS 2003: Proceedings of the 16th International Conference on Neural Information Processing Systems, 8-13 December 2003, pp. 321-328.

[128] Zhou, T., Lü, L. & Zhang, Y.-C. (2009): *Predicting Missing Links via Local Information.* The European Physical Journal B 71/4, pp. 623–630.

[129] Zitnik, M., Agrawal, M. & Leskovec, J. (2018): *Modeling Polypharmacy Side Effects with Graph Convolutional Networks.* Bioinformatics 34/13, pp. 457-466.

[130] Zub, D. (2021): *Scrape Google Scholar with Python.* Available at: `https://dev.to/dmitryzub/scrape-google-scholar-with-python-32oh`, last accessed on 20 July 2022.

Affidavit

„Ich versichere, dass ich die vorstehende Arbeit selbstständig und ohne fremde Hilfe angefertigt und mich anderer als der im beigefügten Verzeichnis angegebenen Hilfsmittel nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen übernommen wurden, sind als solche kenntlich gemacht. Alle Internetquellen sind der Arbeit beigefügt. Des Weiteren versichere ich, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und dass die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht."

Place, date	Signature