Hochschule für
Wirtschaft und Recht Berlin
Berlin School of Economics and Law

Master Thesis

Multi-node representational learning for classifying line-item subgraphs inside order documents

Faculty 1 Economics

Department of Business and Economics

First Supervisor: Prof. Markus Löcher | markus.loecher@hwr-berlin.de

Second Supervisor: Dr. Fabian Brosig | fabian.brosig@workist.com

Berlin, July 7, 2023

By: Eger, Felix | Student ID: 890743 | Koloniestraße 31 | 13359 Berlin

Phone: 0176 31386833 | Email: felix.eger17@gmail.com

Course of Studies: M.Sc. Business Intelligence and Process Management

Eidesstaatliche Erklärung

Hiermit erkläre ich Eides statt, dass ich dir vorliegende Arbeit selbstständig und ohne unerlaubte fremde Hilfe angefertigt, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Berlin, July 7, 2023

MORITZ DEECKE

Abstract

Within the scope of this thesis, table structures of administrative documents are learned via the usage of Graph Convolutional Neural Networks. In particular, tables and their content are represented as nodes in a graph with the task of identifying line item structures of nodes. In total, three different prediction tasks were formulated and trained: straight-odd line item prediction, link prediction, and line item number prediction. Each task was trained on real-life order documents provided by Workist and evaluated on a separate test set. Overall, the link-prediction task significantly outperformed the other two with a Micro F1 score of 90%, compared to a Micro F1 of 78% for straight-odd and 73% for line-item prediction. While the different target tasks proved that learning subgraph structures over node-level representations is generally feasible, the process of node labeling needs to be pointed out for their success. The thesis proves that representing administrative documents as graphs for either node or link predictions is a valid method that can help to conclude subgraph structures present in the graph. Since real-life data was used, the findings are robust and applicable to any order document. While the task was explicitly formulated in document understanding or table structure recognition, its success also motivates subgraph learning in other domains.

KEYWORDS

Graph Convolutional Neural Networks, Graph Neural Networks, Graphs, Subgraphs, Node Prediction, Link Prediction, Table Structure Recognition, Node Labeling, Order Documents, Workist

Table of Contents

1 INTRODUCTION

Document understanding is a field that combines a multitude of disciplines, such as machine learning, natural language processing, and computer vision. To recognize, extract and understand information from a single document is a challenging task that has been an active field of research in recent years.

While the process of recognizing characters from an image can be performed by Optical Character Recognition (OCR), as shown by Smith, 2007, is only a small step in understanding the content of a document. To understand the written content, the usage of Natural Language Processing (NLP) methods, such as Devlin et al., 2018 has proven to dramatically improve the understanding of natural language, while identifying specific objects in documents is usually performed via Object Detection, as shown by Girshick, 2015. The complexity of a single document is manifold - due to variations in format and layout, a mixture of graphical elements and text, and in combination with images and tables. Moreover, as explained by Aiello et al., 2002, authors usually follow a structuring of thoughts and arrange the information accordingly in the document, emphasizing that a lot of hierarchical and relational information between the individual elements is contained in the spatial arrangement. Independent of the method and task chosen to perform intelligent document processing, the market and research has been growing significantly in recent years. **empty citation** estimated the market to reach $\$4.8$ billion by 2022.

A company that challenges the complex task of document understanding is Workist. Founded in 2019, the company reinvents the field of B2B transactions by extracting relevant information from business documents into a machine-readable format. The company enables seamless integration and information exchange between various parties, such as customers, distributors, and clients. The company provides true business value by extracting the written characters, or tokens, analyzing the underlying meaning of tokens concerning their function in the document, and accurately summarising characters into logical units. At the heart of the document understanding process lies, among other technologies, a Graph Convolutional Neural Network (GCN) that captures the meaning of characters in the document.

As the majority of documents used in B2B transactions contain tables that are rich in informative content, the task of identifying and extracting the content from such tables plays a vital role in enabling a seamless extraction flow. Table detection has been an active field of research for many years and has

proven achievable via Object Detection methods, as shown by L. Gao et al., 2017, Saha et al., 2019, and Bhatt et al., 2021, or graph-related methods, Qasim et al., n.d., Riba et al., 2019 and Holeček et al., 2019. While many papers achieve good results in identifying tables as a whole, few focus on the internal table structure and arrangement of tokens inside, which is the thesis's focus.

1.1 RESEARCH OBJECTIVE

Since table structures are grid-like, incorporating rows and columns, the information provided internally is arranged in horizontal and vertical domains. As e Silva et al., 2006 explained, logical connections behind the linear visual clues identify a table. This relationship means that, for example, the information provided in a horizontal line of a table can be seen as a group of combined elements that may or may not be hierarchically analyzed. In business documents, the tokens presented inside such documents can describe a very different part of an order or purchase, for example, the quantity, unit price, currency, or description of an article. While classifying each token into a class that depicts its function, such as unit price, currency, description, and more, is necessary, it does not yet capture the relation between the tokens. Tokens inside an order table can be summarized into separate line items, specifying an entry in order, invoice, or delivery slip, along with its quantities, prices, and additional information. To capture line items as isolated, logical order units are essential to process the information later on. An example business document with marked line items and tokens is shown in **Figure X.**

The difficulty here is that line items combine various aspects of an order. Also, the listed elements vary dramatically from document to document and are only recognizable due to their repetitive nature and logical connection. Whereas table structure recognition has been proven achievable through object detection models, such as Schreiber et al., 2017 or Smock et al., 2022, inferring line items based on graph structures directly has been relatively unexamined. To challenge this task, the question leading this paper is formulated as:

RQ: *"How can the architecture of Graph Neural Networks be modified to classify line-item subgraph structures inside order documents?"*

To identify line items, documents are represented as graphs to capture the relationship between individual tokens and are treated as the only information available for inferring line item structures. The

paper will, therefore, not incorporate any object detection method available to process the information on an image level.

1.2 THESIS OUTLINE

The theoretical foundations discussed in the following chapter will provide a deeper understanding of the field of document understanding, the task addressed, and the model infrastructure used. The Chapter, Theoretical Frameworks, expands on the introduction to Graph Neural Networks (GNN) and Graph Convolutional Neural Networks (GCN) in the literature review and focuses on critical parts of the implemented model and specific research dealing with graph models in document understanding. **Chapter X.** explains the data used for training, its pre-processing and engineering, and the methodology used for training. **Chapter X.** describes the artifacts generated through this study. Following **Chapter X.**, the model performance is evaluated on unseen test data and feature importance analyzed via an ablation study. The results and implications are reviewed and discussed in **Chapter X.**, whereas **Chapter X.** summarises the insights of the thesis and provides and outlook for future research of GCNs in the context of document understanding.

2 LITERATURE REVIEW

This chapter aims to give an overview of the research in Graph Neural Networks in recent years, the domain of table processing, and its combination. At first, the field of Table understanding, necessary prior research, and definitions are explained. The following chapter reviews the data structure of graphs and its potentials and limitations for deep learning. Subsequently, the development of Graph Neural Networks is explained while focusing on critical historical discoveries and recent advancements. Finally, specific papers dealing with table and document extraction based on graph models are reviewed and evaluated.

2.1 TABLE UNDERSTANDING

Inside documents, tables present a structured information organization and likely contain data with some form of logical connection in a horizontal or vertical domain. As defined by e Silva et al., 2006, tables are a graphical grid-like matrix M_{ij}, where:

1. each element i, j of the matrix is atomic

2. there exist linear visual clues, meaning the elements in each row tend to be horizontally aligned
3. linear visual clues describe logical information
4. eventual line art does not add meaning that is not available otherwise

As motivated by the same paper, companies worldwide must process structured documents, such as financial reports. This process can be extended to other departments that process administrative documents, such as procurement, legal, or sales. Especially in procurement documents, such as invoices, order documents, or delivery notes, tables usually contain highly relevant information that needs further processing through an Enterprise-Resource Planning system (ERP).

As evidenced by the sheer number of structured documents processed, the automatic extraction and understanding of tables and their content are vital for many companies. As defined by Coüasnon and Lemaitre, 2014, table processing can roughly be split into three main categories: detection, classification, and recognition. The order of these different tasks expands in difficulty.

As explained by Coüasnon and Lemaitre, 2014, table detection deals with identifying if a table is present in the document and, if yes, where it is placed. Various approaches have been dealing with table detection, such as Riba et al., 2019, Schreiber et al., 2017, and Shafait and Smith, 2010. At the time of writing, Object Detection models such as Schreiber et al., 2017, Smock et al., 2022, and Prasad et al., 2020, seem to be the most prominent model for detecting tables and structures inside of tables.

Table classification, on the other hand, deals with the task of assigning each table or object in a document to a pre-defined class. While it is less common to classify tables as a whole, there are papers that deal with classifying table headers, such as Fang et al., 2012.

Finally, the task of recognizing table structure aims to extract the table's content and infer the function of elements inside the table. As defined by Chi et al., n.d., it does so by recognizing the internal structure of a table. The focus of this thesis can be classified as being part of table structure recognition since it deals with a particular set of table structures - line items. Compared to research on table structure recognition based on object detection, such as Schreiber et al., 2017, or Smock et al., 2022, tables are represented as graphs in the scope of this thesis. This representation means that each table and its content are considered individual objects with some form of relation to each other instead of

the mere image of the document to infer its structure. While this approach has the obvious advantage of directly representing table contents or tokens in the model, it does incorporate some drawbacks for processing graphs as an input data structure. The following chapter explains graphs in computer science and their complexity as an input source for machine learning.

2.2 GRAPHS IN INFORMATION SYSTEMS

Graphs in computer science present a flexible modeling tool, able to capture the relationship between entities and can thus be applied to various tasks in the real world. In the focus of this paper, graphs are used as a data structure. As Rishi Pal Singh, n.d. explains, graphs used as a data structure must be rich enough to mirror the relationships in the real world, and the structure should be simple enough for processing. Both criteria are fulfilled in the application of graphs to the problem of this thesis.

To formally denote a graph G, the definition from Majeed and Rauf, 2020 is lent, which defines a graph G as a set of nodes (or vertices) and edges, connecting any two or multiple nodes within a graph. Mathematically, a graph can be denoted as $G = N, E$, where N is a set of nodes, and E is a set of vertices connecting the nodes. Generally speaking, graphs can either be directed or undirected, implicating a direction between nodes or no direction at all (an undirected graph can be understood as a directed graph with edges between each pair of nodes). An example directed graph is shown in **Figure X.**, in which arrows indicate the direction, and undirected in **Figure X.**. What object the graph, nodes, and relationship between the nodes represent depends on the real-life state or system a graph tries to model. Moreover, edges in a graph can store weights, or as Majeed and Rauf, 2020 refer to as valued or non-valued representing the strength between two nodes. Despite their appealing visual representation, graphs must have an alternative representation for computer processing. As Singh and Sharma, 2012 explain, graphs are best expressed through matrices.

A possible representation of a graph is those of an adjacency matrix. As defined by Singh and Sharma, 2012 an adjacency matrix A represents which nodes are adjacent to which other nodes. For N nodes present in a finite graph G, the adjacency matrix is $A = N * N$, where each entry $a_{ij} = \{1, \text{if there is an edge, 0 otherwise}\}$. The diagonal, therefore, always represents the potential edge pointing back to the node itself, a self-loop. Based on this circumstance, it can be stated that the adjacency matrix will always be symmetric in the case of an undirected graph. An example graph,

including the corresponding adjacency matrix, is shown in **Figure X.**. Another critical metric in graph theory is the degree of a node. Kasiviswanathan et al., 2013 define the degree of a node $deg_n(G)$ for node $n \varepsilon N$ as the count of connected nodes. This definition of a node degree is in the case of an undirected graph. In the case of a directed graph, the degree is separated into the in-degree, the count of incoming nodes, and the out-degree, the count of outgoing nodes for a node $n \varepsilon N$. In the case of an undirected graph, the degree can easily be stored in a degree matrix $D = N * N$, which is filled with zeros except for the diagonal, indicating the count of connected nodes. An example graph and degree matrix are shown in **Figure X.**.

There exist many more details to formally denote the properties of a graph and the operations that can be performed on a graph, such as Depth-first search (DPF), Breadth-first search (BFS), or Minimum Spanning Trees (MST). However, a base understanding of the property of graphs is sufficient knowledge for the operations performed inside Graph Neural or Graph Convolutional Neural Networks.

The flexibility of graphs as a modeling tool enabled a variety of research to use graphs to depict real-life objects, entities, or systems. For example, Girvan and Newman, 2002 used graphs to model social networks, Kleinberg, 1999 modeled the World Wide Web, Newman, 2003 showed the usage of graphs in biology, including protein-protein interactions, and Hagmann et al., 2008 for modeling human brain connectivity. While the paper mentioned above only shows a glimpse of graphs' applicability to model real-life objects or systems, it proves how impressively diverse and powerful graphs can be. Whereas the flexibility of graphs is immense, the irregularity of their structure can be challenging when trying to use them as input data for learning in graph domains. As explained by Battaglia et al., 2018, an explicit representation and an appropriate form of processing are needed to use a learning algorithm for computing interactions between entities and their relations. While images are already considered unstructured data, presenting an image through a graph adds another layer of complexity, the irregularity of its structure to it. Since graphs may have a variable size of unordered nodes and the number of neighbors is not fixed, Wu et al., 2020 explain that essential operations, such as convolutions, cannot simply be extended from an image to a graph domain. This circumstance led to the development of Graph Neural Networks and their successors, explained in the following chapter.

The following should be answered in this chapter:

- What is a graph/ (cover directed, undirected, degree, adjacency matrix)
- Graph theory in Computer Science
- Applications of graphs in real-life
- Introduction to the difficulty and motivation of GCNs

2.3 GRAPH NEURAL NETWORKS

Gori et al., 2005, proposed the first graph neural network architecture in their paper 'A New Model for Learning in Graph Domains'. The authors propose a neural network architecture that enables the usage of graphs as direct inputs to machine learning by extending the prevalent usage of Recurrent Neural Networks (RNN) from that time. Instead of converting a graph to a vector of input representations, the graph can be used directly for node and graph-focused problems, making the architecture robust to handle most input graphs.

Given a graph $G = (N, E)$, where N is a set of nodes, and E a set of edges connecting the nodes N, each node that describes a real-life object can store information in a vector $x_n \epsilon R^s$ referred to as a *state*. However, since nodes can be connected via an edge to neighboring nodes, x_n is naturally defined by the neighborhood of n. To update the vector x_n, each vector is passed through a *transition function* f_w that incorporates the neighboring states of each node. Iteratively, the states of each node are updated based on their state and those of their neighbors. Once the state at point x_{i+1} has a meager difference from the state at time x_i, concluded that it has reached a *stable state*. The weights learned from the neural network, updated during training, are used to determine the impact on the state, and are part of the function f_w. While the paper by Gori et al., 2005 proposes the first GNN architecture, Scarselli et al., 2008 provide a more detailed description of its architecture. While the two papers by Gori et al., 2005 and Scarselli et al., 2008 are milestones in the development of GNNs, the implementation, however, suffers some drawbacks. As Kipf and Welling, 2016 explain, iteratively updating node states in the hope that a stable state will eventually be reached is problematic since there is no guarantee that this will come into effect. In addition, the training and model architecture is very complex, challenging to implement in practice, and computationally expensive.

In difference to the neural network architecture proposed by Gori et al., 2005, Kipf and Welling, 2016 simplify the process of passing messages between the nodes significantly by introducing the concept

of convolutions from Convolutional Neural Networks (CNN), effectively creating a Graph Convolutional Neural Network (GCN). The reasoning for this approach is simple. Traditional convolutions, f.e. applied to an image, slide over parts of the image with a filter to learn local hierarchies. Since graphs do not necessarily follow a grid-like structure, a different approach is needed to capture the local surrounding of a node. The layer-wise propagation rule of nodes to implement the behavior described above, can be denoted as:

$$H^{l+1} = \sigma(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(l)}W^(l)) \tag{1}$$

This rather complex propagation rule can be broken down into steps that explain the base message-passing algorithm in a Graph Convolutional Neural Network.

At first, assuming an input graph $G = (N, E)$ again described through nodes N and edges E has d node features for each node. A feature can effectively be any information that describes the node and play an essential role in a node's informative value. Node features can thus be represented as a feature matrix $X = N * d$. An adjacency matrix A represents a graph with the size of $A = N * N$. If A_{ij} has a value of 1, it indicates an edge from i to j and 0 otherwise. As explained by Kipf and Welling, 2016, self-connections are added to the adjacency matrix A by adding the identity matrix I: $\tilde{A} = A + I_n$, effectively filling the diagonal matrix with ones. Following this step, $\tilde{A}$ is normalized by multiplying the modified adjacency matrix with the degree matrix D. As explained above, the degree of a node is the number of edges connecting to other nodes, stored as D_{ij} with the diagonal indicating the degree of each node. The degree is normalized by taking the inverse square root to prevent higher-degree nodes from dominating the influence on the node features. To preserve symmetry in the matrix and account for the degree of the node sending the message, the inverse, degree matrix is multiplied twice, so that: $\tilde{A} = \tilde{D}^{-1/2} * \tilde{A} * \tilde{D}^{-1/2}$. To complete the layer-wise propagation rule, $\tilde{A}$ is multiplied with the node feature matrix H and a trainable weight matrix W. By wrapping an activation function, such as ReLU, around this part of the equation, a new node feature representation for each node is generated: $H^{l+1} = \sigma(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(l)}W^(l))$.

In the implementation described above, a convolution operation is strictly performed over a 1-hop neighborhood, meaning the direct neighbors of a node n_i for $n \in N$. This aspect of the locality around a node is similar to that of a CNN for image classification. As shown by Wu et al., 2020 a filter applied to a patch of pixel values of an image is essentially a form of capturing the local

neighborhood around the center node. Similarly, the implementation by Kipf and Welling, 2016 is a form of spatial-based graph convolution, as the representation of a node $n \in N$ is updated with the local neighborhood information from nodes around the center node n_i. Spatial methods have a relatively straightforward translation from convolutions on images to the graph domain and have been implemented in various papers, such as Veličković et al., 2017, Monti et al., 2017, H. Gao et al., 2018, and many more. In contrast to spatial methods, an alternative approach to convolution operations, spectral methods, is based on graph signal processing.

Spectral methods represent graphs using the graph Laplacian matrix, as shown by Wu et al., 2020 and Defferrard et al., 2016:

$$L = I - D^{-\frac{1}{2}} * A * D^{-\frac{1}{2}} \tag{2}$$

Here, I_n represents the identity matrix, $D^{-\frac{1}{2}}$, the normalized, inverse degree matrix, and A, the adjacency matrix of graph G, assuming an undirected graph. In this case, L always contains real numbers and is symmetric, allowing for further graph-related operations. Assuming a graph signal x_i, which is a feature vector holding information about the graph nodes, a Graph Fourier Transform can be applied. Similarly to audio signal processing, in which the Fourier transform decomposes an incoming signal into sine waves, the Graph Fourier Transform decomposes the graph signal into eigenvectors of the Laplacian L. As shown by Wu et al., 2020, the inverse of the Graph Fourier Transform is an exact representation of the incoming signals within the frequency domain, allowing for easier convolutions. A filter that is then applied to the inverse allows the transformation of the graph signals back to the node level while extracting important local structures - similar to the filter of a CNN. As Wu et al., 2020 explain and evident by the implication of the filter, the choice of filter is a significant driver for the expressiveness of the extracted structures.

An important graph convolutional network that uses a spectral method is the paper introduced by Defferrard et al., 2016. Its advantage lies in the definition of the filter applied. The filter is strictly localized and can control how far, in terms of nodes from the center node, the filter expands. To do so, Defferrard et al., 2016 derive $\tilde{L}$, which is a transformation of the above-described Graph Laplacian, such that:

$$\tilde{L} = 2 * \frac{L}{\gamma_{max}} - I \tag{3}$$

Here γ_{max} denotes the largest eigenvalue of L and I, the identity matrix. Then, using Chebyshev

polynomials, the filter size can be controlled via the parameter K. The Chebyshev polynomials can formally be denoted as:

$$T_k(\tilde{L}) = 2 * \tilde{L} * T_{k-1}(\tilde{L}) \tag{4}$$

In this case $T_0(\tilde{L}) = I$ and $T_1(\tilde{L}) = \tilde{L}$. Finally, by applying a filter $g(\theta)$, a new representation is learned by a simple multiplication:

$$g(\theta) = \sum_{k=0}^{K-1} \theta_k * T_k(\tilde{L}) \tag{5}$$

θ represents parameters learned during training, similar to weights in a neural network. As visible, K expands the filter convolution by reaching k-hops nodes away from the center node. The paper of Kipf and Welling, 2016 offers a simplified propagation rule than the one proposed by Defferrard et al., 2016. Nonetheless, both are excellent implementations of convolutional operations on graphs, while one is spatial- and the other a graph-based method.

While the convolutional operations on graphs, as described above, are still used today, they have some drawbacks concerning the nature of transductive learning. Since the node embeddings and weight matrix learned in the paper of Kipf and Welling, 2016 depend on the presence of all nodes within the graph, extending the learned weights to a scenario in which nodes are not presented during training is generally challenging. As Hamilton et al., 2017 explain, this is a big drawback in domains where input graphs evolve frequently, and unseen nodes might be encountered during testing. To alleviate this problem, the authors propose Graph Sample and AggregatE, GraphSAGE, a model capable of inductive learning. Instead of learning a weight matrix, node features are leveraged to learn an embedding function that can be applied to new nodes and unseen graph structures. As explained in the paper, three main steps are performed inside GRaphSAGE. At first, a fixed-size set of neighborhoods is sampled around each node in the graph, reducing the computational footprint within a batch. Next, an aggregator function is wrapped around the features of the neighborhood. Which aggregation function to use might depend on the task. As specified by Hamilton et al., 2017, a mean aggregation, Long Short Term Memory (LSTM) aggregation, or pooling approach could be applied in this scenario, whereas the authors decided to use a max-pooling. Finally, the aggregated neighborhood is concatenated with the node's features and fed through a fully-connected layer to generate a new embedding. The learned function is transferable between nodes as the weights are shared across nodes. A common

denominator in each of the papers above is the fact that the neighborhood of a node is incorporated into the node features, either by a convolution operation, Kipf and Welling, 2016, Defferrard et al., 2016 or by learning an embedding function, Hamilton et al., 2017. Inspired by the advancements of neural machine translation, Veličković et al., 2017 propose to extend the ability to deal with variable-sized inputs to the domain of graphs. By being able to focus on the most relevant parts of an input, the concept of self-attention is introduced, enabling the model to compute the representation of a single sequence. The idea is to learn a hidden state for each node based on its neighbors and a self-attention mechanism. The proposed architecture offers three main advancements. As the operation is computationally efficient, and also offers inductive learning, as in the paper of Hamilton et al., 2017, it offers a promising architecture. At first the node features are fed through a linear transformation. Following this step, the attention scores are calculated, measuring the importance of a node to another. Across each node neighborhood, the attention scores are normalized using a softmax function to attain a representation ranging between 0-1. The normalized attention scores are then used to compute a weighted feature matrix of each node's neighbors based on the attention scores. After being passed through a RElU activation function, the output features are obtained. The model learns independent attention-weighted functions and representations by employing multi-headed attention, meaning several different attention mechanisms and weights. The final representations are then concatenated and form the final embeddings.

While each model architecture explained above learns node embeddings or embedding functions differently, each method can be applied to similar downstream tasks. The three most prominent prediction tasks are node classification, edge classification or link prediction, and graph classification, Wu et al., 2020. While each paper mentioned above implements some form of node prediction, only Gori et al., 2005, Scarselli et al., 2008, and Defferrard et al., 2016 implemented graph classification. The task of link prediction will be introduced in the later **Chapter X.**.

The flexibility of graphs as an input structure to represent real-life relationships and entities enables a variety of use cases based on Graph Neural Networks. The paper by Kipf and Welling, 2016 first applied GCNs to a citation network to identify similar classes in a network, which can be seen as a form of social network analysis. Veličković et al., 2017, and Chen et al., 2020 used the same datasets, whereas the latter also extended the usage to other social network datasets, such as YouTube,

Flickr and BlogCatalog. Kearnes et al., 2016 applied GCNs to predict the molecular fingerprints based on molecular structure, whereas Veličković et al., 2017 tested their GAT model on protein-protein interaction. Apart from molecular chemistry, Ying et al., 2018 trained a GCN to generate node embeddings based on 3 billion pins and boards from Pinterest to build a high-quality recommendation engine, outperforming the current deep-learning architectures. Y. Li et al., 2017 use a graph model for traffic forecasting, while Yao et al., 2019 demonstrate the ability of GCNs for text classification.

In the scope of this paper, GCNs will be used for document understanding. There have been numerous attempts by Riba et al., 2019, Qasim et al., n.d., Holeček et al., 2019, Chi et al., n.d. and Lohani et al., n.d. to use graph-based models for different document-related tasks. Workist, the partner company of this paper and provider of the data, exemplifies the usage of GCNs in practice to process complex documents, such as invoices, purchase orders, delivery orders, and many more. Within the document service offered by Workist, GCNs play a vital role in classifying document tokens to enable scalable, robust and error-free B2B-transaction processing.

The following should be answered in this chapter:

- Introduction to neural networks and motivation for them
- Architecture of a base GCN and differences to normal neural networks
- Use cases of GCNs
- Latest developments, must successful architectures

2.4 GRAPH NEURAL NETWORKS IN DOCUMENT UNDERSTANDING

There have been few attempts to use GNNs or GCns to understand documents, each with a slightly different focus. As explained before, valuable data is presented inside a table in many industries. Recognizing a table and extracting its content presents a valuable task to prevent the manual processing of documents. To challenge this task, Riba et al., 2019 represent documents as a graph with nodes capturing blocks of text and edges as spatial relationships between them. By training a GNN, they can classify whether a node is part of a table. Similar to this approach,Holeček et al., 2019 focus on tables and their content. However, by extracting features from input images and feeding through a pipeline of CNNs and LSTM networks, they can extract the table's relationship between rows and columns. A similar approach is taken by Qasim et al., n.d., who treat document pages as graphs, with

text lines as blocks and edges representing spatial relationships. By doing so, they can classify nodes into categories and infer hierarchical relationships between them. Chi et al., n.d., dives deeper into recognizing complicated table structures like nested cells spanning multiple rows. They can extract information from the documents and recognize the internal table structure by proposing a two-step process of detecting cells via an R-CNN and the recognition based on a graph-based method. In contrast, Lohani et al., n.d. focus less on identifying tables and their content but on extracting key-value pairs from inside the document to extract complex patterns inside the document. While each paper has a different focus and uses different data, they present a valuable basis for understanding challenges and opportunities in graph-based representations to understand documents.

The common ground for each of the methodologies described above is that each uses some form of graph-based representation to understand the layout or content of the document. Lohani et al., n.d., Riba et al., 2019, Holeček et al., 2019, Chi et al., n.d., and Qasim et al., n.d., each uses a method to infer edges based on spatial relations between the nodes. Edges are an essential aspect for capturing tabular structures in the document, since, as Riba et al., 2019 point out, they offer a complementary dimension to the raw textual data. Riba et al., 2019, Lohani et al., n.d., Holeček et al., 2019, Qasim et al., n.d. all create nodes based on individual words, whereas Chi et al., n.d. create nodes based on cells, which could potentially contain multiple words separated by a white-space. While each of the paper state they are utilizing some form of spatial relationships to infer edges between each token, only Lohani et al., n.d. and Chi et al., n.d. point out their method for doing so. To reduce the number of edges to a reasonable size, Chi et al., n.d. infer edges between each token based on K nearest neighbors (KNN) method. In difference to this approach, Lohani et al., n.d. use a graph modeling algorithm that attributes words as part of horizontal lines, then reads words line by line and derives edges based on the vertical projection. While this approach is the best documented and easily reconstructable, it also poses some further advantages, as discussed in **Chapter**. In conclusion, no matter which method is chosen, creating relations in the graph based on the spatial relations of the document is a well-accepted method.

While each method transforms its documents into some graph-like representation, the networks trained are very different. Qasim et al., n.d. and Riba et al., 2019 use a Graph Neural Network to classify nodes into different categories or as part of the table, respectively. While not explained in detail,

Chi et al., n.d. use a graph-based method that is likely similar to that of a Graph Neural Network. In contrast, Lohani et al., n.d. make use of a GCN, in detail the variant utilising Chebyshev polynomials as described in **Chapter X**. Finally, Holeček et al., 2019 utilize a pipeline consisting of a CNN and LSTM to model the relationships between rows and columns. Since each method used is very different and part of a multi-step pipeline, it seems logical that the chosen network for learning the graph representation needs to be carefully evaluated depending on the use case.

The dataset used for training needs to be carefully evaluated to understand the implications of the results. Riba et al., 2019 utilize the CON-ANONYM and the RVL-CDIP dataset, containing of 960 and 518 images, respectively. Both datasets come from industrial and administrative sources, hand-labeled by the authors, and scanned by ABBYY OCR. In comparison, Lohani et al., n.d. use a private dataset with 3100 invoices provided by a private company, hand-annotated by the authors at the word level. Tesseract OCR was the chosen tool for text extraction. In comparison, Holeček et al., 2019 used a much bigger dataset containing 4848 and 35880 PDF pages. The documents are of various vendors and layouts and annotated by hand and an algorithm. The tool used for extracting text from inside bounding boxes was not disclosed. Similarly, Chi et al., n.d. also created their dataset, SciTSR, for the task, containing 15000 tables in PDF format. The tables are created based on LaTeX source files, which are interpreted to infer the position of tables and text inside the document. Lastly, Qasim et al., n.d. also generated tables and content based on HTML source files to create a large dataset consisting of 500000 tables in total. Summarising the datasets used in each of the papers, the applicability of each result is hardly transferable between them since they are trained on specific documents and types. While some papers, such as Riba et al., 2019 use a small dataset, limiting the general expressiveness of their model to unseen data, Chi et al., n.d. and Riba et al., 2019 automatically generate the datasets based on source files.While the automatic creation of training data overcomes the issue of a small dataset, it creates the issue that the model will likely be very biased to the specific input document type and not be transferable to a real-life problem in which administrative documents might be created from all sorts of sources. Artificial datasets are a vital restriction when analyzing each paper's applicability since understanding structured layouts becomes much easier if the source of creation always follows the same rules and layouts. In the case of Chi et al., n.d., it is likely that the trained model performs much better on LaTeX source files (the source of training data), while it might struggle with unseen

document sources, such as Word, Excel, or documents coming from ERP systems.

Whereas the data sources are very different, many of the critical problems pointed out in the paper are similar. As Chi et al., n.d. point out correctly, the structure of a table can become very complex, and many existing methods cannot cope with this complexity to extract reasonable results - an issue also encountered in this thesis. This problem is repeated by Holeček et al., 2019, stating that invoices usually suffer from very specialized structures. Moreover, document formats vary significantly in their tabular layout, as pointed out by Riba et al., 2019. Therefore, Lohani et al., n.d. emphasize the need for a robust model to adapt to new structures for proper extraction. In addition, each of the papers, not automatically created from source files, needed to be hand-annotated - a significant constraint for creating models that can generalize on larger data volumes. However, even if larger datasets are created automatically from source files, the resulting images suffer from noise, such as different fonts and sizes, and in the case of real-life documents or scans, even physical damage, such as scratches or smudges.

While each paper plays a vital role in improving the task of document understanding and extraction, they also highlight the state and complexity of the current state of research. While some papers have achieved promising results, the trained model is hardly generalizable to data from unseen sources, emphasizing the need for more extensive and encompassing datasets, as well as novel and robust methods enough to capture the nuances and complexity of administrative documents.

The following should be answered in this chapter:

- Graph representation comparison - edge creation and node capturing
- Networks used - Graph Neural Networks
- Datasets
- Complexity of the tables, such as:
- Variation of data
- Sparsity of labels
- Noisy data

3 THEORETICAL FRAMEWORKS

This section highlights fundamental concepts discovered in prior research that were adopted to the specific problem, i.e., line item detection. At first, the task of identifying subgraphs is introduced, after which an essential pre-processing step called node labeling is explained. Finally, the specific problem of line-item detection is discussed based on prior research.

3.1 SUBGRAPH LEARNING VIA GCNS

There have been very few attempts to explicitly learn subgraph representations, partly based on the difficulty of representing subgraphs. A graph must be formally defined to clarify these circumstances. Based on the definition by Wang and Zhang, 2022, let $G = (N, E, X)$ be a graph described through a finite node set N, a set of edges E, and a node feature matrix X, whose i^{th} row describes the feature of node i. A subgraph $S = (N_s, E_s, X_s)$ is a subgraph of G if $N_s \subseteq N$ and $E_s \subseteq (N_s * N_s) \cap E$ and X_s is a stack of rows that correspond to the nodes within the subgraph. In essence, this definition points out that every node in the subgraph S must also be part of the larger graph G, while every edge in the subgraph S must also be part of the larger graph G. Since each node is described through a row in the feature matrix, the rows corresponding to the nodes that are part of S correspond to the feature matrix X_s. Applying this definition to the use-case of this thesis, it becomes evident that each line item that is part of a document graph automatically qualifies as a subgraph in the larger document graph G. While this is not a surprising fact, diving deeper into the task of predicting subgraph structures reveals several challenges. Alsentzer et al., 2020 point out four main issues in the scope of subgraph learning:

1. **Varying Structure and Size**: Subgraphs are not necessarily a repeatable pattern that is easily identified within the main graph. Instead, they can be spread out, vary in size, and be placed far apart. Alternating sizes and structures also occur in line items since they do not necessarily encompass a single line but several ones, with alternating amounts of tokens. The question is how to represent these subgraphs effectively.
2. **Internal and External Connectivity**: Subgraphs are seldom isolated structures but connect to nodes outside the subgraph. Since message passing inside a GNN is performed based on the edges, external connectivity will also influence subgraph representation - raising the question

of how to account for external connections.

3. **Location inside the larger Graph**: Subgraphs could be situated anywhere inside the larger graph - either concentrated in one area or spaced apart. Learning the correct position and the effect on predicting subgraphs is crucial.
4. **Shared edges and nodes**: Subgraphs likely share the same edges or non-edges, making subgraphs dependent on each other. Since the data in the scope of this thesis is reduced to line-item tokens only, subgraphs are always connected, increasing the difficulty of distinguishing them.

In their paper, Alsentzer et al., 2020 presented the first novel approach to tackle the task of subgraph learning - SUBGNN. The training is initialized by setting an anchor patch, A_i, which is a set of nodes from the corresponding subgraph S_i that is adjacent to but not inside the subgraph. Following this step, each node has features assigned to them. The main innovation then lies in a three-channel message passing that deals with the issues mentioned above of subgraph learning. The first channel corresponds to internal messages that capture the subgraph and its structure. Incoming messages, the second channel, are computed based on messages passed from the anchor patch into nodes inside the subgraph. Finally, outgoing messages, the third channel, capture messages passed from inside the subgraph to nodes in the anchor patch. Each channel is separately aggregated and then concatenated before passing it through a neural network layer to update the node features based on a learnable weight matrix. While this proves to help learn subgraph representations, it is not fit for the problem at hand since it does not perform the task of subgraph detection but subgraph representational learning. Subgraph representational learning is similar to the task of graph prediction, in which representations of whole graphs are computed, just with the modification that the representation refers to a subgraph within a larger graph.

However, a few aspects, as pointed out by Wang and Zhang, 2022 make the usage of SUBGNN difficult in practice. At first, the three-channel message passing proposed is computationally expensive and challenging to implement. Moreover, the proposed approach highly depends on the initial definition of anchor patches. In a task such as line-item prediction, where the position and number of line items are unknown prior, the implementation becomes difficult in practice. Instead, Wang and Zhang, 2022 propose a different operation, a node labeling propagation step instead of the three-channel message passing. The usage of such node labels has been proposed in various papers, such as Wang and

Zhang, 2022, Zhang et al., 2021, Zhang and Chen, n.d., P. Li et al., n.d. and Jiaxuan You et al., n.d.

The theoretical implications and different implementations of Node Labelling are discussed in the following chapter. The node labeling trick used by Wang and Zhang, 2022 implies an easier-to-implement and more accurate subgraph representation. The label propagation layer starts with assigning a label to each node based on its position or the relation within the subgraph. Following this step, one message passing round is performed, collecting and aggregating information from the neighbor (whose convolution operation is up to the user). Finally, an activation function is wrapped around the updated feature vector to ensure the values stay within a suitable range. Afterward, the message passing of choice is performed, while further label propagation steps could be deployed later in the model. This modified architecture, GNN with Labeling Tricks for Subgraph Representation Learning (GLASS), outperforms most of the baselines and performances of SUBGNN.

3.2 NODE LABELLING

The concept of labeling nodes has been used in various studies and has proven useful when used correctly in the scope of the problem. An example from Zhang et al., 2021 is introduced to motivate the usage of node labeling. Assuming a graph as in **Figure X.**., where nodes (v_1, v_2) and (v_3, v_4) are isomorphic. Graph isomorphism, as defined by McKay et al., 1981 occurs for two graphs, G_1 and G_2, if a one-to-one mapping exists that enables to map the nodes of G_1 perfectly onto G_2, such that the adjacency of two nodes in G_1 is perfectly adjacent to those of G_2. In easier words, graph isomorphism occurs when every node in the first graph has an exact counterpart in the second graph and vice versa. A node-pair in the first graph connected by an edge must always have an exact counter-pair in the second graph. While the example shown in **Figure X.**., does not have two graphs, it does, however, have two perfectly isomorphic subgraphs such that without the labels, it would be impossible to distinguish between the node pair (v_1, v_2) and (v_3, v_4). As Zhang et al., 2021 conclude, this becomes a severe problem in many graph-related tasks, such as link prediction. When performing several hops to incorporate neighboring attributes and update the node features, nodes v_1 and v_3 would have the exact representation due to the similarity of the neighborhood. For example, predicting which node is more likely to link with (v_4) would become difficult in this scenario. The abstract example proposed can be directly transferred to the problem of line item classification since many line items are repetitive and isomorphic by nature. Therefore, distinguishing a node in line-item x from a node

in line-item $x - 1$ might become extremely difficult when the local neighborhoods are practically identical. To mitigate this issue, node labeling forces each node to have unique characteristics that enable the graph to distinguish one node from the other.

One of the most straightforward node labeling steps, proposed by Zhang et al., 2021 involves applying a one-hot label for nodes that are part of a subgraph S. By doing so, representations are learned at a subgraph, instead of a node-level, which performs better than Alsentzer et al., 2020. Depending on the prediction task, the node labeling is adjusted accordingly. Zhang et al., 2021 propose a labeling trick that assigns the same label to the two target nodes (for which a link is supposed to be predicted) and iteratively assigns larger nodes the further they are from the two center nodes. This way, the structural information of the enclosing subgraph is incorporated into the nodes. In comparison, P. Li et al., n.d. incorporate the distance between nodes to create unique node features. The distance is calculated based on a set of reference nodes. The distance could be based on a shortest path or diffusion-based method. In either scenario, the node feature will provide additional context based on the relationship to other nodes. Jiaxuan You et al., n.d. instead use a form of node coloring to encode unique identities to distinguish the nodes from each other. While node labels are either assigned based on attributes information, such as Zhang et al., 2021 or position P. Li et al., n.d., Jiaxuan You et al., n.d. focus on assigning unique identities to each node. Incorporating a one-hot vector into the node features distinguishes the nodes from each other. This approach is quite different as the one-hot label carries no semantic information. Since node labeling proved a successful method to overcome graph isomorphism, it is adopted in the scope of this paper and added as a node feature. The exact details are explained in **Chapter X**.

3.3 LINE ITEM DETECTION

Lastly, some more papers need to be addressed to narrow in on line item prediction. While there have been many advancements in document analysis via object detection algorithms, Schreiber et al., 2017 or Smock et al., 2022, the task of analyzing documents based on graph methods is relatively sparse. In particular, line item detection is rarely the sole focus of a paper. The only two papers implicitly dealing with line-item detection are: 'Table understanding in structured documents' Holeček et al., 2019 and 'Complicated Table Structure Recognition' Chi et al., n.d. While the focus of both papers has been explained in **Chapter X.**, the task of line-items detection has not explicitly been investigated.

Holeček et al., 2019 use the whole document as input and try to classify nodes. One of these categories is line items, which are represented inside a table in the document. While training the model on a smaller dataset and achieving a promising result of an F1 score of 93%, it is unclear whether the aspect of line-item detection tried to identify separated line items and their enumeration as labels or label all tokens as line items inside the table body. Nonetheless, the result seems promising and shows that the task of line-item detection is possible. Chi et al., n.d., on the other hand, focused on more complex table structures, The evaluation of models on complex tables achieved a Micro F1 score of $72,5\%$, which is decent but highlights the issue of detecting line items spanning several lines in a table. In this thesis's scope, line items usually span multiple lines and do not necessarily follow a regular grid-like structure. Therefore, Chi et al., n.d. results seem more relevant to the proposed problem. However, the data used by Chi et al., n.d. is based on LaTeX-source files making it easier for a model to understand formatted tables than tables created by different ERP systems, as in the case of this study.

4 METHODOLOGY

The methodology aims at explaining what decisions were made during training, why they were made, and what outcomes are expected. The data will be introduced at first, including all necessary pre-processing steps and the different target labels used during training. Following this explanation are the modeling methodology, training, and evaluation methods.

4.1 DATA

This section elaborates on all data used, processing, and creating features for individual nodes in a graph. At first, the data is analyzed and described to capture the complexity and nuances of it. Following the introduction, the creation of edges, nodes, and edge features and the different prediction tasks are explained.

4.1.1 DESCRIPTION

The data used in this study consists of real-life order documents used for processing orders between B2B transactions of small-to-medium-sized enterprises across many European markets and the United States. This fact already highlights that document languages can vary between English, French, Spanish, German, and many others. Moreover, the documents are created based on different ERP systems and are not standardized based on the same source. The pages inside a document might also be copied

and skewed and include noise from smudges, tilting, or poor resolution. Each document is split into individual pages, and treated as a single instance, meaning one page equals one graph as an input to the model. Workist provides all of the data, and due to privacy regulations, only demo documents are displayed in this paper.

Since the data is processed along a pipeline inside a document understanding service of Workist, the training documents already contain valuable information before modeling. At first, each document is analyzed via an Optical Character Recognition tool (OCR) that detects tokens and their position on the document. A token is any character, letter, or digit containing informative content; a white space separates that. The concept of tokens is essential since one coherent text might be separated into several tokens. For example, the text *steel bolts* would be separated into the tokens *steel* and *bolts*. Furthermore, the OCR tool performs the task of identifying the position in the form of a bounding box coordinate $bbox = [coord_{left}, coord_{top}, coord_{right}, coord_{bottom}]$ relative to the page's width and height and converting the character into a string containing the content.

Following the OCR step, the tokens are transformed into a graph $G = N, E$, where N are the tokens identified prior by the OCR and E edges between the tokens that are created based upon a methodology by Lohani et al., n.d., that is explained in the following chapter. The resulting graph is used as an input to a Graph Convolutional Neural network that classifies each node into a category, which explains its function within the document. In total, there exist over 70 different categories. Example categories are *OrderLineItemSupplierArticleNumber*, *OrderLineItemDescription*, *OrderLineItemCurrency*, and more. Finally, each page is converted into an image and used as input into an Object Detection Model that identifies essential areas on the page. These are, for example, a table, a header, and individual line items. The classified tokens can now be assigned to each line item based on the overlap of the bounding box. The last part is to be replaced by a Graph Convolutional Neural Network. An example document of the graph can be seen in **Figure X.** In total, there are X amounts of documents available before pre-processing.

Continue with Grammarly

4.1.2 DATA PRE-PROCESSING

Each document is converted into a graph with tokens as nodes to enable the learning of graph representations of documents. For each node, the available information is stored as node features. The

same amount of features are available for every node and stored in a matrix $X = N * d$, with N being the number of nodes and d the number of features. Each document is different in its layout and not limited in the number of line items. Since line-item counts are not fixed, they result in significantly imbalanced line-item counts within the dataset, as visible in the Histogram in **Figure X.** At the start, each document contains all tokens, even those not part of the table and, consequently, the line item. To reduce the complexity of the predictions, noisy tokens not part of a line item are excluded. A comparison with a document before and after filtering out line-item tokens is visible in **Figure X.**. This pre-processing step simplifies the training for the model, as it reduces the random noise around the relevant line item tokens by eradicating all tokens that are not part of it. This step is feasible in a real-life scenario since the token classification model should already identify the relevant line items. Indeed, the model performance would be impacted by any earlier miss-classification of the token classification model. However, this is an acceptable risk outside of identifying line items. Consequently, this step immediately drops all documents without line items since training and testing will not be necessary. In addition, all documents containing only a single line item are dropped. Even though this step might seem irrational and suffers from a form of data snooping, as this information would not be available to the model beforehand, it will be argued in **Chapter X.** that this information can be inferred by another labeling trick performed later. Finally, these pre-processing steps result in X graphs for training, testing, and evaluation.

4.1.2.1 EDGE CREATION

As explained, nodes within the graph are the tokens detected by the OCR tool and labeled as part of a class. The question of how to connect the nodes within each graph plays a central role as it directly influences the message-passing algorithm of the trained model. The idea of generating edges is based on the proposed implementation by Lohani et al., n.d. In the paper, the author describes an algorithm to infer a grid-like representation of edges between the tokens by connecting each token to the closest neighboring tokens in a horizontal and vertical dimension. This approach offers numerous improvements compared to a brute-force approach that would connect each token available tokens in the document. At first, the resulting graphs are more computationally efficient, as they significantly reduce the number of edges. Secondly, it limits the degree of each node to four edges, and it infers some form of ordering since nodes can be understood in a horizontal or vertical ordering. This

ordering mirrors the arrangement of words in the document and the writing in natural language, as shown in **Figure X.**, traversing from left to right would correctly result in the text ... tbd

4.1.3 NODE FEATURES

Node features used as an informative source for training are based on the content of the token. Since the node features represent individual tokens and are frequently updated through neighboring nodes, they play a vital part in the model's ability to learn how to represent nodes within the graph. A single feature vector for a node is based on several data sources. Each feature is explained through its calculation and explained below:

4.1.3.1 NORMALISED BOUNDING BOX COORDINATES

Since each token stored within a node is fenced through a bounding box rectangle, each node has bounding box coordinates available as spatial features. The bounding box coordinates are stored based on their top, left, right and bottom coordinates. Since non-normalized features can cause deep learning models to suffer from vanishing and exploding gradients during backpropagation, bounding box coordinates are normalized based on a document page's width and height:

$$tbd \tag{6}$$

4.1.3.2 RELATIVE POSITION

The bounding box coordinates are compared to the table coordinates to capture the relative position of each token's bounding box within a page. Therefore, the bounding box centroids are calculated for the X and Y coordinates:

$$X_{centroid} = \frac{X_{left} + X_{right}}{2} \tag{7}$$

$$Y_{centroid} = \frac{Y_{top} + Y_{bottom}}{2} \tag{8}$$

The table width and height are merely the difference between the right/left and bottom/top coordinates, respectively:

$$Table_{width} = X_{right} - X_{left} \tag{9}$$

$$Table_{height} = Y_{bottom} - Y_{height} \tag{10}$$

The relative X and Y coordinates for a node are calculated as follows:

$$X_{relative} = \frac{X_{centroid} - Table_{left}}{Table_{width}} \tag{11}$$

$$Y_{relative} = \frac{Y_{centroid-Table_{top}}}{Table_{height}} \tag{12}$$

By doing so, each node receives relative X and Y coordinates within the relative space of the table area.

4.1.3.3 WORD INDICES

Since the word indices are enumerated integers based on the token classification model, they must be normalized to prevent exploding gradients during model training. Word indices are normalized using the min and max index, like the bounding box coordinates. For each word index, we apply the following transformation:

$$Index_{normalised} = \frac{Index - Min_{index}}{Max_{index} - Min_{index}} \tag{13}$$

4.1.3.4 LABEL ENCODINGS

Since the nodes were classified prior by a token classification model, each node had a label assigned to it. Overall, there were over 60 labels present. Example labels are: $OrderCustomerNumber$, $OrderInvoiceAdressEmail$, $OrderProjectNumber$. Node features are processed into node labels using scikit-learn's LabelEncoder, and the resulting integer labels are one-hot encoded using scikit-learn's OneHot Encoder. Finally, each label was represented by a 72-dimensional vector.

4.1.3.5 BETWEENESS

The betweenness centrality of a node is a graph metric that captures the node's centrality within the graph. In mathematical terms, it is the sum of the fraction of all shortest paths for a node pair (s, t) that pass through the node v. If N is the set of nodes of a graph G, then the betweenness centrality of node v is denoted as:

$$B(v) = \sum_{s,t \in N} \frac{\sigma(s, t|v)}{\sigma(s, t)} \tag{14}$$

, where s and t are all other nodes in the network.

4.1.3.6 EDGE

Edge is a one-hot encoded feature vector, indicating for the possible four coordinates (left, top, right, bottom) whether a neighbor exists in that direction. In mathematical terms, $edge = (e_{left}, e_{top}, e_{right}, e_{bottom})$

is a four-dimensional vector, where for each direction i:

$$e_i = \begin{cases} 1, & \text{if edge exists.} \\ 0, & \text{otherwise.} \end{cases} \tag{15}$$

4.1.3.7 DISTANCES

Similarly to the above vector, distances is a feature vector indicating the distance calculated by the edge creation method of Lohani et al., n.d. Therefore, the equation is adopted to be $distance = (d_{left}, d_{top}, d_{right}, d_{bottom})$ so that for each direction i:

$$d_i = \begin{cases} distance_{edge}, & \text{if edge exists.} \\ 0, & \text{otherwise.} \end{cases} \tag{16}$$

4.1.3.8 HORIZONTAL EDGE

Each token can have two horizontal edges (e_{left}, e_{right}), whereas the token to the left or right could be of the same token class as the source node. The node feature Horizontal Edge, is therefore a two-dimensional vector (e_{left}, e_{right}) where fore each direction i:

$$e_i = \begin{cases} 1, & \text{if target node is of same token class.} \\ 0, & \text{otherwise.} \end{cases} \tag{17}$$

4.1.3.9 VERTICAL EDGE

Analogous to the above feature, each token can have two vertical edges (e_{top}, e_{bottom}). The node feature Vertical Edge is, therefore, a two-dimensional vector (e_{top}, e_{bottom}) where fore each direction i:

$$e_i = \begin{cases} 1, & \text{if target node is of same token class.} \\ 0, & \text{otherwise.} \end{cases} \tag{18}$$

4.1.3.10 DEGREE

The final feature, the degree of the node, is a one-dimensional vector. As explained above in **Chapter X.**, the degree in an undirected graph represents the number of edges for a node i. Based on the edge creation method is chosen, each node can, at max, have four edges in the coordinates, so summing up the edges gives the total amount of edges, $sum_{edge} = sum(e_{left}, e_{top}, e_{right}, e_{bottom})$, where for each

edge:

$$e_i = \begin{cases} 1, & \text{if edge exists.} \\ 0, & \text{otherwise.} \end{cases} \tag{19}$$

4.1.3.11 NODE COLOURING

As explained in **Chapter X.**, labeling nodes as information in the node features before training the model can significantly boost the performance. Due to the issue of graph isomorphism and the non-trivial enumeration of line items from a node perspective, as explained in **Chapter X.**, a node labeling trick was used to boost the learning. The node labeling in this thesis is based on node coloring, which associates nodes with color, indicating the potential line item. While it is hard to assign such colors for many nodes, after all, the problem of line item detection is the use case, it is relatively easy to identify candidate nodes as root nodes within each line item. Since each document is based on a Latin language, read from left to right and top to bottom, line items consequently originate on the left side. In addition, line items follow repeatable patterns - though not necessarily alike from the content, the columns in a table ensure consistency among the vertical domain. Based on these aspects, axioms about the creation of tables are defined:

1. **Table Root Node**: Within each document graph $G = (N, E)$ with N being a set of nodes and E a set of edges, a list of top-left tuples $C = (top, left)_1, (top, left)_2, ..., (top, left)_3$ can be created, storing the top-left-coordinates for each node in the graph. When calculating the min of this list, $L_{min} = min(C)$, the tuple minimizing the function can be extracted. Since line items are always enumerated from to i for i line items present and read from left to right and top to bottom, the smallest top-left tuple, indicating the node on the top left, must always be part of the first line item. This node is called a root node.
2. **Parallelism**: Since line items, even in the case of spanning cells in the horizontal and vertical direction, always follow a grid-like representation, line items in a table must always run parallel.
3. **Horizontal Connectivity**: The aspect of parallelism implies that any pair of tokens (n_1, n_2) that are part of N and align horizontally must always be part of the same line-item. If this were not the case, line items could cross each other and not follow a grid-like structure.
4. **Root Node Connectivity**: In addition to a horizontal grid-like structure, tables have a vertical

structure. For each document graph G, the nodes of one line item form a subgraph S_i, with i being the line item number, so that $S_i \subseteq G$. Due to the condition mentioned above, an existing subgraph S_{i+1} must be placed below for any subgraph S_i. Otherwise line item two could come after three, which would not follow a logical enumeration. This enumeration implies that for each node $n_i \in S_{i+1}$, the top-left tuples of all nodes in S_{i+1} must be below the top-left tuples of all nodes in S_i. Therefore, when traversing the graph vertically starting from the root node in S_1, all other subgraphs in G will eventually be encountered.

5. **Root Node Classes**: Concluding the axiom of Root Node Connectivity, it is important to consider the labels L assigned to each node in N. Since each node has a label assigned to it, given by the token classification model and indicating the function inside the table, a final axiom can be stated. The successful identification of the root node n_1 and its label l_1, determine the root for the first subgraph S_1 in G. Since each subgraph S_i describes a line-item that is placed in a grid-like structure, each line-item must contain a node $n_i \neq n_1$ which has the same token label. Since the labels are arranged based on their function in a columnar layout, the additional root nodes with the same token label must be encountered when traversing the graph vertically and indicate the next root node of a subgraph $S_i \neq S_1$.

Figure X. visually shows the implementation of these axioms. Furthermore, the axioms lay the groundwork for a node feature encapsulating global line item structures on a node level. A recursive algorithm was developed based on the axioms, which performs the node coloring and is described in the pseudo-code below:

Algorithm 1 Graph Traversal

Require: A graph $G = (N, E, X)$, with N being a set of nodes and E a set of edges. X is a feature matrix $X = d * N$, with d being the amount of features, including the bounding box coordinates of a token.

Initialize empty root node and empty root token class
for each node in N **do**
    Store the node bounding box top and left values
end for
Minimize over the sum of stored bounding box values to update root node and class
Initiate list of candidate root nodes
Call **Algorithm 2: Explore-candidates-recursively** using bottom neighbor of root node
for each node in candidate root nodes **do**
    Call **Algorithm 3: Determine-most-left-node** with left neighbor of candidate root node
end for
for updated left nodes in candidate root nodes **do**
    if left node label is equal to root token class and has a right neighbor **then**
        Append most left node to root node
        Assign new line-item class to root node
    end if
end for
for each root node in push lists **do**
    Call **Algorithm 4: Horizontal-recursive-run** with right neighbor of root node
end for

Algorithm 2 Explore-candidates-recursively

Require: Bottom node b

if $b \neq$ None **then**
    Assign b to list of candidate root nodes
    Call **Algorithm 2: Explore-candidates-recursively** using bottom neighbor of b
end if

Algorithm 3 Determine-most-left-node

Require: Left node l

1: **if** $l \neq$ None **then**

2: Call **Algorithm 3: Determine-most-left-node** with left neighbor of l

3: **else**

4: **return** l

5: **end if**

Algorithm 4 Horizontal-recursive-run

Require: Right node r

1: **if** $r \neq$ None **then**

2: Assign class label to node r

3: Append node r to push list

4: Call **Algorithm 4: Horizontal-recursive-run** with right neighbor of r

5: **end if**

The algorithm performs four main steps. At first, it determines the root node by minimizing the top-left tuples of each node. Then it traverses the graph vertically by exploring all the root node's bottom neighbors. Each neighbor with the same token class as the root node is deemed a root node for another line item. By traversing horizontally to the right neighbors of each root node, all neighbors that are part of the same line item are found.

4.1.4 EDGE FEATURES

Since the token are connected via edges, the distance between the bounding box of two connected tokens can be computed and added as a distance attribute to each edge. The distances are used as the basis for determining an edge feature:

4.1.4.1 NORMALISED, INVERSE DISTANCES

Distances between bounding boxes are normalized as a basis for edge weights. At first, for a given document graph $G = (N, E)$, with N being a set of nodes and E a set of edges and d_i denoting the

distance for edge i, the inverse distance of each edge $e \in E$ was calculated by:

$$d\_inv_i = \begin{cases} \frac{1.0}{d_i}, & \text{if } d_i \neq 0. \\ 0, & \text{otherwise.} \end{cases} \tag{20}$$

Next maximum and minimum inverse distances are found:

$$id_{max} = max(d\_inv_1, d\_inv_2, ..., d\_inv_i) \tag{21}$$

$$id_{min} = min(d\_inv_1, d\_inv_2, ..., d\_inv_i) \tag{22}$$

Finally, the normalized, inverse distances nid_i are calculated:

$$nid_i = \frac{id_i - id_{min}}{id_{max} - id_{min} + \epsilon} \tag{23}$$

, in which ϵ denotes a tiny constant to prevent division of zero error. The computation is essentially a form of min-max-normalization to ensure that the distances within each document graph can only range between $(0, 1)$

4.1.5 TARGET LABELS AND TRAINING TASKS

The predicted labels are compared against different true labels to calculate the loss of the model after each iteration and update the weights. Different true labels mean that different targets and approaches are used to model the task to infer line item structures:

4.1.5.1 BINARY STRAIGHT ODD PREDICTION

The first node prediction task used binary labels as true labels for each node. Even though the number of line items per document varied greatly and consequently the number of line-item classes, each node could be classified as being on a straight or an odd line item. When counting the line items available from top to bottom, each node can be classified as being on either a straight or an odd line item. Per each document, each node had a binary label assigned to it, making it a binary classification task. Even though this approach does not predict the line-item number directly, it has interesting properties to make it a more feasible prediction task. At first, assigning binary labels to the nodes will, in most cases, solve the issue of a highly imbalanced dataset. As explained in **Chapter X.**, the amount of line items per page is highly imbalanced across all pages. When using binary labels, the imbalance for pages with an odd amount of line items decreases with every line item added, as visible in **Figure X.**.

Additionally, there is a more nuanced reason why binary labels are preferable over enumerated ones. The task is a node prediction task, and passing messages in the graph between nodes creates weights and node embeddings used to infer a label. As graphs do not naturally have a direction, it is unclear from the node point-of-view which node should have a higher label than another. Since line items usually look similar, including the same arrangement, token classes, and content, inferring an order from this information is counterintuitive if the graph does not contain a direction. Especially in the case of many line items on a page assigning a large numerical value, f.e. 60, and a smaller one, f.e. 2, based on comparable node embeddings, is a difficult task even for a human. While this problem is not solved in the binary classification case, it is mitigated since the logic of alternating zeros, and ones is simpler than enumerated numbers.

4.1.5.2 LINK PREDICTION

To dig deeper into the complexity described above, consider the following task. Assuming, when presented with the nodes and information in **Figure X.**., the task is to infer the line-item number. While this task is purposefully set up to be confusing for the reader and does not accurately meet the complexity of a message-passing algorithm, it does, however, highlight the central issue of inferring an enumeration. The task becomes easy once a picture of the whole page is presented, as in **Figure X.**.However, purely basing the idea on the information on a node level is rather tricky - even for a human. This aspect is the reasoning for the second target, link prediction, which solves the issue by not considering any numerical ordering at all. The notion for this approach is based on the idea that after a few convolutional rounds, the network will eventually create meaningful node embeddings between nodes on the same line item. While it is still difficult to correctly attribute these node embeddings to an integer value, comparing pairs of nodes for their similarity of node embeddings becomes easier. Node similarity is calculated through the cosine similarity between each pair of nodes connected through an edge and compared against the true value. Similar node embeddings should achieve a score closer to 1, implying that the edge correctly connects a node-pair on the same line item, while a score closer to -1 implies an incorrect edge between a node-pair not on the same line item. While this task does not directly identify line items, it does help rule out incorrect edges in the hopes of creating disconnected subgraphs for each line item. The simplicity of this approach, paired with the interpretability, makes it a handy approach, as proven later during the evaluation of results.

4.1.5.3 LINE ITEM PREDICTION

Finally, the last task available to train is the extension of the binary prediction of **Chapter X.** by predicting the line-item number directly. Effectively, the class imbalances described in **Chapter X.** are accepted by necessity. While the issues with this approach have been pointed out in-depth in the previous two chapters, it is helpful to tackle this challenge nonetheless, at least for reasons of comparability. Using this approach also implies that a biased assumption is made toward the number of line items. Since a fix-sized last layer is needed to predict line item classes, the total number of line items across all pages must be known prior. While this information is easy to extract from the available data, it might not represent reality accurately since documents with higher line item counts are likely to exist in reality. The trained model, however, will always be limited to the fixed size layer.

4.2 MODELLING METHODOLOGY

The target labels described beforehand lay the groundwork for approaching the task through three different methods: straight-odd line item prediction, link prediction, or line item prediction. Since each method approaches the task of predicting line items from a slightly different perspective, the training and evaluation also differ. Overall, the graph architecture chosen for learning the line item representations is the Graph Convolutional Network with Chebyshev Polynomials proposed by Defferrard et al., 2016. The reason for the choice is motivated by the paper Lohani et al., n.d., since the edge creation method of this paper was adopted, it achieved promising results in a comparable node classification task in documents. Additionally, it is likely that for predicting subgraph structures on a node level, spectral methods can capture more information about the surrounding local neighborhood than a spatial method.

For each training task, a separate model was trained in Python 3.9.15. The model architectures are based on the PyTorch library 1.12.1, with the models loaded from torch-geometric 2.2.0. For training Workist provided one Nvidia Tesla M60 8GB GPU.

4.2.1 TRAINING METHODOLOGY

The three targets are trained separately on the same data before evaluating the resulting models on an unseen test set. Optimal parameters are determined based on hyperparameter tuning used for the models in the final evaluation. The data used for training was processed such that each document

available for training contained at least two line items. The motivation and implications for this step were explained in **Chapter X.** and will be discussed again in **Chapter X.** Each model's weights are initialized based on the Xavier/Glorot Initialisation explained in Glorot and Bengio, 2010. Neural network weights and biases are randomly initialized. However, choosing randomization and its scale is crucial since it can lead to exploding or vanishing gradients during backpropagation. Therefore, Glorot and Bengio, 2010 propose a weight initialization based on a normal distribution around the center 0, which effectively creates random weights that are more symmetric, preventing the issues above during backpropagation. The optimizer chosen for the three tasks is Adam, with a weight decay set for the learning rate. The choice of Adam is motivated in part by comparable papers dealing with the task of document understanding, such as Lohani et al., n.d. and Holeček et al., 2019. The weight decay is a form of regularization that adds a penalty to the loss function, such that the modified loss L' is calculated by:

$$L' = L + \frac{1}{2}\lambda\|\mathbf{w}\|^2 \tag{24}$$

Here λ denotes the weight decay, and $\|\mathbf{w}\|^2$ is the squared L2 norm of the weight vector. The exact choice of learning rate and weight decay is explained in **Chapter X.**. A step learning rate scheduler was implemented with a step size of 20 and a gamma of 0.1. After 20 epochs, the learning rate is multiplied by gamma, forcing the learning rate to decay over time. The loss function for the two node classification tasks (straight-odd line items, enumerated line items) is the negative log-likelihood loss. For the link-prediction case, binary cross-entropy with logits loss was chosen. The negative log-likelihood loss is a standard loss function for multi-classification tasks and well-suited for the multi-label classification task when predicting all line-item numbers. Since each node in the graph exclusively belongs to one label, negative loge-likelihood is preferable as it predicts out-labels separately for each token. Due to this condition, the loss function was also adopted for the straight-odd prediction task. Regarding the link-prediction task, the input values for the loss function are slightly different from the node classification task. As explained, the results of the cosine-similarity scores between a node pair in the graph cause the resulting scores to range between -1 and 1. Within the binary cross entropy with logits loss, the values are first passed through a sigmoid activation function, mapping them to values between 0 and 1. The resulting values are optimal for applying a cross-entropy loss to measure the difference between predicted and true labels. A dropout ratio was

chosen after each convolutional layer in the model architecture to prevent the model from overfitting. Dropout is a very effective yet simple measure to prevent overfitting and was proposed by Srivastava et al., 2014. By randomly picking a certain amount of output nodes that are ignored for predictions, the model becomes less sensitive to specific weights. The choice of a, the ratio of picked nodes, is a measure for controlling the number of nodes dropped. The parameter a was tuned, and the optimal value was explained in the following chapter. Gradient Clipping is an additional measure to prevent exploding gradients, first introduced by Pascanu et al., 2013. Gradients are scaled back to a threshold to prevent them from becoming overly large. In the scope of this thesis, the chosen threshold was set to 0.5. The overall training time and computational overhead are reduced as the model is constantly evaluated on validation and hold-our set during training. None of the data contained in the training is part of the validation set. None of the data contained in the training is part of the validation set. The validation set serves two purposes. At first, it enables performance tracking on unseen data, while training is the basis for an early stopping mechanism. The mechanism stores the best performance of the model on the validation set at epoch i, and updates it in case there is an improvement. The training is aborted if no improvement occurs after x epochs since the model does not improve. In training, early stopping was set to a size of 20.

4.2.2 TUNING METHODOLOGY

Within the training, there is a multitude of parameters to be tuned to optimize the model performance. While it is not feasible to train all parameters in combination, a subset of relevant parameters was selected for tuning. The list of parameters are: Batch Size, Dropout Ratio, Hidden Neurons, Learning Rate, Weight Decay, Amount of Convolutional Layers, K (for Chebyshev Convolutional Layer). The exact values to be chosen for each parameter are shown in Table 1.

A random search was chosen for tuning, as it has been proven more effective than a grid search, Bergstra and Bengio, 2012. Since training runs on the complete data available are time-consuming, the number of searches was limited to 25 and performed twice, once for the link-prediction task and once for the straight-odd prediction task. As the prediction task and loss function is the same, the results of the straight-odd prediction task are also chosen for the direct line-item prediction task. The exact results of each tuning run, including the optimal parameters, are shown in the following chapter. Plenty of other relevant tuning parameters were likely neglected during tuning. However, this thesis

focused on showing how to model the task. The same random state was set for each tuning run, creating the same train-test splits. The overall tuning time took roughly 240 hours.

4.2.3 EVALUATION METHODOLOGY

The best-performing parameters identified by the tuning are used for the final models to test the performance. Each final model was trained three times with a separate random state, and the results were averaged to minimize the effect of a sampling bias. Each model was tested on unseen test data created from the same random state to prevent data leakage. Accuracy, Micro and Macro F1, precision, and recall are calculated to evaluate the model performance. The precision, recall, and F1 score are also reported for the individual prediction classes of each task to enable a deeper understanding of the direction of errors in the model.

The accuracy in a classification case is the proportion of correctly classified nodes/edges over the whole amount of nodes/edges predicted during training or testing:

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}} \tag{25}$$

While it is a useful standard metric, it does not generate enough insights and can be misleading in the case of imbalanced data sets. To understand the F1 score, precision, and recall must first be explained for the training task. Precision is the proportion of true positive predictions among all positive predictions. In the case of the straight-odd case, a positive prediction implies an odd line item and a correct edge in the link-prediction scenario. It shows the model's ability to classify relevant instances correctly. In the cases of binary node prediction, it is the ability to classify odd line items correctly or to classify edges connecting nodes from the same-line item correctly:

$$\text{Precision}(P) = \frac{\text{True Positives}(TP)}{\text{True Positives}(TP) + \text{False Positives}(FP)} \tag{26}$$

The recall, however, is the proportion of true positive predictions among all actual positives, implying how well the model can identify all relevant instances:

$$\text{Recall}(R) = \frac{\text{True Positives}(TP)}{\text{True Positives}(TP) + \text{False Negatives}(FN)} \tag{27}$$

The F1 score is then the harmonic mean between the two, balancing the result of each inside a single number:

$$F1 = 2 * \frac{\text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}} \tag{28}$$

In difference to the above calculation, the Micro F1 score calculates the total true positives (TP), false negatives (FN), and false positives (FP) for each class and therefore captures a potential class imbalance:

$$\text{Micro F1} = 2 * \frac{\sum_i TP_i}{\sum_i TP_i + 0.5 * (\sum_i FP_i + \sum_i FN_i)} \tag{29}$$

The Macro F1 score calculates the results for each class separately and then averages the results, meaning it does not account for class imbalance:

$$\text{Macro F1} = \frac{1}{n} \sum_{i=1}^{n} F1_i \tag{30}$$

In the scope of the data used, both metrics are useful. Micro F1 is essential, especially for the case of directly predicting line items, as it accounts for class imbalances, while the Macro F1 is helpful as each line item is equally important, independent of the class balance in the data.

5 ARTIFACT DESCRIPTION

This chapter focuses on describing the artifacts that were generated within the scope of this study. In total, three different types of models were trained, tuned, and tested that are available for further usage. The first model, was trained for a node classification task, by assigning binary labels (straight-odd) to each node. The second model was trained to predict whether each edge in a graph correctly connects a node pair on the same line-item. Lastly, another node classification model was trained, aimed at assigning specific class labels for each line-item. Each model was based the architecture of Chebyshev convolutional layers, by Defferrard et al., 2016.

5.1 Model 1: Straight-odd prediction

The first artifact created is a trained, tuned, and tested model designed to predict a binary label $(0, 1)$ for each node in the graph. A zero indicates a straight, a one an odd line item. Based on the pre-processing of the data explained in **Chapter X.**, it is assumed that each input graph only contains line-item tokens. As explained above, the architecture used for training the model was based on Defferrard et al., 2016, as it has proven to work successfully in a related document understanding task by Lohani et al., n.d. Additionally, since the edge creation method by Lohani et al., n.d. was adopted for each input graph, the resulting graphs had a similar edge frequency for each node. The best-performing parameters for the binary classification case are documented in Table 3. Additionally, the

parallel coordinate chart provided in Figure 3 indicates the most relevant parameters of the successful training runs, reinforced by the parameter importance concerning the F1 score in Figure 4. While this task does not explicitly predict line-items, it will be argued later in **Chapter X.**, that it should be relatively easy to infer the enumeration in case straight-odd labels are predicted.

5.2 Model 2: Link prediction

The second artifact is another trained, tuned, and tested model designed to predict a binary label $(0, 1)$ for each edge in the graph. A 0 indicates an incorrect edge, i.e., connecting a node pair of different line items, while a 1 connects a node pair on the same line item. As evident by the tuning results presented in Table 2, this task performs much better than any of the other artifacts. Similarly, as before, the parallel coordinate chart in Figure 1 and the parameter importance in Figure 2, indicate the most relevant parameter for the model. While this task does not predict line items at all, it does serve the purpose of modifying the graph as a structure to isolate line item subgraphs. By deleting irrelevant edges and keeping relevant ones, the input graph is ideally pruned to S isolated subgraphs for S line items in the table. While it would require an extensive post-processing step to convert the output graph into specific line-item predictions, it will be argued later in **Chapter X.** that the artifact should instead be used as a pre-processing step in a pipeline then a single end-to-end model.

5.3 Model 3: Line item prediction

The last artifact created in this chapter is a trained and tested model designed to assign integer labels to each node, $1 - x$ for x line-items in the graph. Due to time constraints, the model was not tuned. Instead, the hyperparameters from the straight-odd node classification task were adopted. While it is hard to estimate whether the hyperparameters are ideal for the model, it is likely to be the option, as it is the same prediction task, loss function, and model architecture used by Defferrard et al., 2016. Since enumerating line items across each document creates a heavily imbalanced dataset, the artifact is likely not the best approach for the task of predicting line items, which is discussed further in the the following chapter.

6 EVALUATION OF RESULTS

6.1 PERFORMANCE ON UNSEEN DATA

Overall 44646 document graphs are used for training, testing, and evaluation. These documents are pre-processed, as explained in **Chapter X.**, meaning they do not include documents without line item tokens or documents with only a single line item. 20% of the data is reserved for testing, i.e., 8929 document graphs. The same amount of data is used for validating the model's performance during training. Therefore, each model was trained on 26788 documents in total. Each training task was evaluated three times with a different train-test-split to reduce a potential sampling bias. Therefore, the performance results presented are averaged across three training runs, with different random states for splitting the data. The standard deviation of each run is also reported to capture any deviations between the runs.

6.1.1 Model 1: Straight-odd prediction

The performance of the first model, predicting straight-odd line items, is presented in Table 4. The first row presents the model performance on all the data available. On average, the model has a 78% accuracy and Micro F1 score, with a slightly lower Macro F1 score. While it might seem alarming that the precision and recall are equal, this is a normal behavior in a binary prediction case. Since micro averaging the results globally by summing the per-class true and false positive counts will always be the same. Therefore the accuracy is also equal to the Micro F1 score.

The data was split into buckets and evaluated separately to highlight better and worse-performing document cases. The rows represent the buckets, Easy, Medium, and Hard Graphs. An easy graph is considered a graph in which a line item does not extend a single line, creating a near-perfect grid, similar to an Excel sheet. Medium graphs, presented in the example **Figure X.**, are document graphs in which line items span up to three lines, i.e., up to three tokens vertically stacked. Finally, a hard graph is any document graph in which line items contain more than three lines in a singe line item, as shown in **Figure X.** The results for the straight-odd prediction task indicate that the model performs much better for easier graphs. In contrast, the performance drops significantly from roughly 84% to 68% Macro F1 score. These results demonstrate that the model struggles with increasing complexities of data but also that the overall level of difficulty of document graphs in the test set

varies dramatically. The difference between the Macro Precision and Recall indicates that the model is slightly more likely to incur False Negatives than False Positives. Lastly, since documents with single line items were excluded in training, though they represent a realistic example encountered in an application scenario, they were added back in a separate test run. The test data results, including the single line items, are shown in the last row, 'All difficulties incl. SLI'. The results show that the performance does not decrease compared to the data excluding single line items in the first row. Instead, the performance is even slightly better than without. This circumstance can be explained by the fact that documents containing a single line item are easily marked via the node coloring method and thus instead increase the amount of correctly labeled nodes.

The direction of error committed by the model can be understood when looking at Table 7. The results show the F1 score, Precision, and Recall, for the individual target labels, i.e., the prediction of a straight or an odd line item. The values are also averaged across three seeds. The results show that the model can predict both target labels decently. Overall, the model performs slightly better at predicting straight line items, which does not come as a surprise, as there are generally more straight line items present. Since there exists no real trade-off between performing better in one than the other class, it is difficult to say which case weighs more in those prediction scenarios.

6.1.2 Model 2: Link prediction

The performance of the link prediction task is shown in Table 5. Overall the performance is much better than the straight-odd prediction task. While the performance on all data is roughly around 90% accuracy and 89% Macro F1 score, the performance drops when switching from easy to hard graphs. The accuracy drops around 4% when going from easy to hard graphs, and the Macro F1 score is around 7%. The results also indicate that the model struggles with more complex documents. However, the performance is not impacted as much as the straight-odd prediction case, suggesting that the model performance is overall more robust. In addition, the standard deviation between accuracy and the F1 scores is much lower than the straight-odd prediction task, indicating the model can easier reproduce the performance even with different train-test splits. When including single line items in the data, the performance stays roughly the same as without, with a slight tendency to have an even better performance.

Looking at the performance concerning individual target labels, it is evident that the performance for

both targets is more than decent. The model performs well, especially in catching the correct edges, i.e., edges that combine nodes on the same line item. In contrast, the model is certainly more prone to misclassify incorrect edges than correct ones, potentially leading to higher false positives of incorrect edges. In the scope of the use case of identifying line items, this trade-off is better than the other way around. Since the task is to detect line items by creating connected subgraph structures, it is arguably worse to disconnect edges between tokens on the same line item than the other way around. While it would be ideal for balancing both cases, the cost of leaving some incorrect vertical edges between line items is a somewhat acceptable drawback.

6.1.3 Model 3: Line item prediction

The results for predicting line item numbers are visible in Table 6. As evident, the performance is lower than the other two training tasks, which is to be expected given the difficulty of up to 73 heavily imbalanced target labels. Interestingly, the performance is not much worse than the straight-odd prediction task, with a Micro and Macro F1 score of roughly 73%. Moreover, the drop in performance between Easy, Medium, and Hard graphs is equally strong compared to the change for straight-odd labels. Same as before, the model performs much better, with a Micro F1 of 79% and Macro F1 of 80%, on easier graphs, compared to more difficult ones, 72% for Micro F1 and 65% Macro F1. Interestingly, the ratio between Macro Precision and Recall behaves contrary to the other two models. Especially for medium and hard graphs, the model likely misses many node instances where a line item is present. Reintroducing single line items causes a much better performance concerning the Macro F1 score averaged across three seeds. This behavior is likely due to the model's ability to detect better the first line item, label 1, than higher-order line items. Since 73 target labels were available for prediction, the model was not evaluated for individual targets.

As the targets in this prediction task were generally more difficult than the other node classification task, straight-odd prediction, the model performed expectedly worse. Nonetheless, the model could learn from the data and achieve a competitive performance against the other node classification task.

6.2 ABLATION

7 DISCUSSION

Modeling tasks I want to discuss that

1. gernerally good results
2. it was proven the task was learnable
3. best performing models did not model the task directly
4. postprocessing needed
5. explain potential to process link-prediction via betweeness score
6. explain potential to perform clustering on binary task
7. show results when using a multi-step pipeline

Within the scope of this thesis, it has been shown that the task of learning node representations to infer line item association is feasible. While the best-performing task, link prediction, did not learn line items directly, predicting straight-odd line items on a node level was also proven possible, yet less performant. Predicting line item numbers directly on a node level suffers from serious complications arising from the imbalance of line item numbers, the uncertainty regarding the maximum amount of line items across all documents, and the difficulty of enumerating node labels.

Different post-processing steps must be discussed to address the results and show that the first two tasks could be used in a production environment. At first, when predicting the links between nodes, the ideal resulting graph should have x amount of isolated subgraphs for x amount of line items in the document graph. As shown in **Figure X.**, the resulting document graph is unlikely to be always correct after cutting edges. As evident, incorrect edges between the line items likely remain. A post-processing step is needed to identify such edges. A possible implementation could utilize the betweenness score of a node, whose implementation details have been discussed in **Chapter X.** In the example provided, the incorrect edge has a betweenness score of **...**, and is ranked **...** highest of all scores in the graph.

Alternatively, when using straight-odd line-item predictions, another post-processing step is possible. Since line items are repetitive, an odd line item must always follow a straight one and vice versa. Therefore, line items could likely be identified using a clustering method. By separating each node into two batches - straight and odd - and then clustering the resulting nodes based on their top coordinates inside the batches, line items are easily separable based on the coordinates. For the document shown in **Figure X.**, a resulting clustering run based on the two batches is shown in **Figure X.**. The

method chosen for the clustering is DBScan, with an epsilon of $epsilon = ..$ and $min\_samples = 1$. While the clustering method is decisive for identifying individual line items, the example certainly motivates further investigation.

Lastly, the individual models trained could be used in a multi-step pipeline. As the input graph for the node-classification model still contains several edges that incorrectly connect nodes between alternating line items, the link-prediction model could be used to pre-process and prune the document graph. By receiving a document graph as an input, pruning incorrect edges, and only keeping relevant ones before passing the result into the straight-odd prediction model, the model will likely be more confident regarding correct line items. **Table X.** shows the resulting runs on a test set that performs the multi-step pipeline, outperforming the base straight-odd prediction case **more discussion once results are there**. Finally, other trivial post-processing steps are reasonable in the scope of a production environment. As pointed out through the axioms in **Chapter X.**, line items can never cross each other. This circumstance also infers that tokens that are horizontally aligned can never have a different line item number because otherwise, the axiom would not hold. Tokens and their predictions should thus be evaluated after the model's prediction and checked whether there are any conflicts on a horizontal line between the line item enumeration. In case a conflict due to alternating line items arises, assigning the highest count of line item numbers to the conflicting nodes is reasonable. In addition, the results should also be enriched based on line item metadata about the page. This information could come from an object detection model, detecting the count of line items beforehand. By cross-validating the results, incorrect predictions can later be corrected. Independent of the exact pre- or post-processing steps implemented, some form of processing seems reasonable when implementing a model with line item predictions in a production environment.

Node Labeling

1. emphasises the need for node labeling in subgraph tasks
2. shows the weakness of gcns to infer information on a node label about global line-item structures
3. explain weakness of the task due to missclassification
4. emphasis that this task is likely to be the most relevant part for improving model performance -> potentially in combination with an object detection task on image basis

While the task of straight-odd and link-prediction was proven to be feasible, it is essential to point out that the most critical node feature with informative value about line items is the node coloring step, explained in **Chapter X.**. The reasoning behind the success of the feature likely points as well to the obstacle of the overall task of predicting line items. Since line items exist on a global document level above the node but below the overall graph level, inferring node-level predictions about these structures is challenging. Even features such as bounding box coordinates do not carry information regarding structures above the node level. This complication is shown **Figure X.**. Only when looking at all bounding box coordinates globally does a pattern emerge which is not available to the nodes at the time of message passing. Since one coordinate in the graph might relate to line item 1 in one document instance and line-item 2 in another, it is almost impossible to assign a line item based on this information confidently. Line item structures appear identifiable when comparing node features globally while looking at the graph as a whole. A complex pre-processing method such as node coloring is necessary to break down this global information onto node-level information, available before the first round of message passing. Because line items inside document graphs are highly isomorphic, it becomes even more challenging to differentiate one from another or by learning node embeddings that could reveal a form of numerical ordering. The link-prediction task likely outperformed the other two since it alleviated the overall burden of inferring the ordering. Even though node labeling boosted the model performance significantly, it is not error-free and likely even provides eventual, incorrect information. As explained, the first axiom explained in **Chapter X.** is only correct in about 96% of the cases. Nonetheless, a form of node labeling implemented to capture global subgraph structures as a node-level feature is a good starting point for further research. A potential approach in future research could combine object detection models to break down subgraph structures into node features for node classification tasks.

Data Preprocessing

1. emphasise the need for excluding single line-items
2. explain the weakness of doing so
3. show how to mitigate this effect
4. potentially run results including single line-items

As explained in **Chapter X.**, the data was pre-processed to fit the modeling task by removing class

imbalances. While this is a crucial step to enable learning in the first place, it does introduce a form of selection bias, as the data used to train the model does not represent the entire document population. Since documents with single line items do frequently occur, there is a need for dealing with such documents. However, introducing single line items into the test data for validation does not alter the performance of any of the three models. Since documents containing single line items are likely easily identified by the node coloring feature, by assigning only a single color to the graph, any graph-related model will likely infer that only a single line item is present. The node coloring method successfully identified a single root node only for the amount of single line item documents, in **percentage** cases. Therefore, the method is highly accurate in identifying such documents, which does not incorporate any form of deep learning at all. While out of the scope of this thesis, it could also be possible to perform a graph-level prediction that predicts the number of line items prior to any node-level predictions by pooling all graph nodes of the final layer into a graph-level representation.

Missing evaluations

1. only includes node or edge level tasks
2. no subgraph representational learning, since predictions needed to be broken down onto node level predictions

Lastly, another potential task was left out but could be helpful in the scope of the use case. As proposed in the paper of Wang and Zhang, 2022 and Alsentzer et al., 2020 it is possible to learn subgraph representations of several nodes, similar to representations of a whole graph. A combination of both approaches would first learn subgraph embeddings and then use such embeddings as features for individual nodes to break down the information of subgraph embeddings to a node level. While the details and the success of such a method are unclear, subgraph embeddings combine both subgraph and node-level features, potentially bridging the missing informative gap between global subgraph structures and node-level informative features. Subgraph embeddings would likely help the model have a higher dimensional representation of how line item subgraphs actually look, rather than a one-dimensional feature vector such as the node coloring. In addition, it would deal with another interesting task of multi-node representational learning, explicitly for subgraphs. By doing so, the different structural representations of line items can be evaluated, and possible ways to use a graph model to achieve similar embeddings for identification purposes. In the scope of this thesis, however,

the task was to identify the subgraphs for individual nodes. Assigning nodes to subgraphs implies that the initial knowledge regarding the subgraph position in the bigger graphs is unavailable, thus not allowing for subgraph representational learning in a production environment.

8 CONCLUSION

Within the scope of this thesis, three different Graph Convolutional Networks were trained to answer the research question: *How can the architecture of Graph Neural Networks be modified to classify line-item subgraph structures inside order documents?"*. It has been shown that predicting links between node pairs is a more feasible task to identify line item structures than to classify nodes directly. Instead of using the final layer to predict integer labels, using node embeddings as input for comparing node similarity seems to abstract the task better and achieve a better performance. While each task has a different performance, it is vital to point out that the choice of initial node features likely greatly impacts the model's ability to predict subgraph structures. The usage of a node labeling trick, as introduced by Zhang et al., 2021, Wang and Zhang, 2022, or Zhang and Chen, n.d., is an efficient method for enabling node-level predictions about subgraph structures. While this method boosts the model performance, it is vital to point out that it is not error-free and likely introduces some form of bias. Moreover, the model training did not incorporate other subgraph representations as node features, which could, in theory, enable an even more informative type of learning. Due to the results and limitations, future research should focus on finding more robust node labeling methods to enable subgraph learning through node classification. While research regarding subgraph representational learning exists, such as Alsentzer et al., 2020 or Wang and Zhang, 2022, little work focuses on predicting existing subgraph structures on a node level. Robust node labeling methods outside the scope of document understanding are likely a promising start to fuel node-level representational learning about subgraph structures. While this thesis only dealt with detecting line item subgraph structures, it can generally be extended to any subgraph structures modeled within graphs. Apart from detecting alternative document subgraph structures, such as headers, address lines, or even complete tables, this prediction task could be applied to practically any data represented via graphs.

Summary of findings: A restatement of your research question, along with a summary of the results and findings. No new information should be introduced here; it should only encapsulate what has already been discussed in your thesis.

Interpretation: Discuss the significance of your results in the context of your research question. This might involve comparing your findings to the original hypotheses and to the findings of previous research in your field.

Impact: Discuss the implications of your work for your field of study. How has your work added to the existing body of knowledge? What impact could it have on future research or on the real-world applications of your study?

Limitations: Acknowledge any limitations of your research. These could include limitations in your methodology, your sample, or in the scope of your study. This part helps to build the credibility of your research by showing you have a thorough understanding of the boundaries of your study.

Future Work: Recommend areas for further research that your work has opened up. These could be specific aspects that you were unable to address in your thesis, or new questions that your research has raised.

Concluding Thoughts: End on a strong note that highlights the importance of your work and leaves a lasting impression on the reader. This can be a comment on the broader implications of your work, or a final insight that you want your reader to take away.

REFERENCES

Aiello, M., Monz, C., Todoran, L., Worring, M., et al. (2002). "Document understanding for a broad class of documents." *International Journal on Document Analysis and Recognition*, *5*1, 1–16.

Alsentzer, E., Finlayson, S., Li, M., & Zitnik, M. (2020). "Subgraph neural networks." *Advances in Neural Information Processing Systems*, *33*, 8017–8029.

Battaglia, P. W., Hamrick, J. B., Bapst, V., Sanchez-Gonzalez, A., Zambaldi, V., Malinowski, M., Tacchetti, A., Raposo, D., Santoro, A., Faulkner, R., et al. (2018). "Relational inductive biases, deep learning, and graph networks." *arXiv preprint arXiv:1806.01261*.

Bergstra, J., & Bengio, Y. (2012). "Random search for hyper-parameter optimization." *Journal of machine learning research*, *13*2.

Bhatt, J., Hashmi, K. A., Afzal, M. Z., & Stricker, D. (2021). "A survey of graphical page object detection with deep neural networks." *Applied Sciences*, *11*12, 5344.

Chen, F., Wang, Y.-C., Wang, B., & Kuo, C.-C. J. (2020). "Graph representation learning: A survey." *APSIPA Transactions on Signal and Information Processing*, *9*, e15.

Chi, Z., Huang, H., Xu, H.-D., Yu, H., Yin, W., & Mao, X.-L. (n.d.). "Complicated table structure recognition."

Coüasnon, B., & Lemaitre, A. (2014). "Recognition of tables and forms".

Defferrard, M., Bresson, X., & Vandergheynst, P. (2016). "Convolutional neural networks on graphs with fast localized spectral filtering." http://arxiv.org/pdf/1606.09375v3

Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2018). "Bert: Pre-training of deep bidirectional transformers for language understanding." *arXiv preprint arXiv:1810.04805*.

e Silva, A. C., Jorge, A. M., & Torgo, L. (2006). "Design of an end-to-end method to extract information from tables." *International Journal of Document Analysis and Recognition (IJDAR)*, *8*2-3, 144–171.

Fang, J., Mitra, P., Tang, Z., & Giles, C. L. (2012). "Table header detection and classification." *Proceedings of the AAAI Conference on Artificial Intelligence*, *26*1, 599–605.

Gao, H., Wang, Z., & Ji, S. (2018). "Large-scale learnable graph convolutional networks." *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, 1416–1424.

Gao, L., Yi, X., Jiang, Z., Hao, L., & Tang, Z. (2017). "Icdar2017 competition on page object detection." *2017 14th IAPR International Conference on Document Analysis and Recognition (ICDAR)*, *1*, 1417–1422.

Girshick, R. (2015). "Fast r-cnn." *Proceedings of the IEEE international conference on computer vision*, 1440–1448.

Girvan, M., & Newman, M. E. (2002). "Community structure in social and biological networks." *Proceedings of the national academy of sciences*, *99*12, 7821–7826.

Glorot, X., & Bengio, Y. (2010). "Understanding the difficulty of training deep feedforward neural networks." *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, 249–256.

Gori, M., Monfardini, G., & Scarselli, F. (2005). "A new model for learning in graph domains." *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, *2*, 729–734.

Hagmann, P., Cammoun, L., Gigandet, X., Meuli, R., Honey, C. J., Wedeen, V. J., & Sporns, O. (2008). "Mapping the structural core of human cerebral cortex." *PLoS biology*, *6*7, e159.

Hamilton, W., Ying, Z., & Leskovec, J. (2017). "Inductive representation learning on large graphs." *Advances in neural information processing systems*, *30*.

Holeček, M., Hoskovec, A., Baudiš, P., & Klinger, P. (2019). "Table understanding in structured documents." *abs 1804 6236*, 158–164. https://doi.org/10.1109/ICDARW.2019.40098

Jiaxuan You, Jonathan M Gomes-Selman, Rex Ying, & Jure Leskovec. (n.d.). "Identity-aware graph neural networks."

Kasiviswanathan, S. P., Nissim, K., Raskhodnikova, S., & Smith, A. (2013). "Analyzing graphs with node differential privacy." *Theory of Cryptography: 10th Theory of Cryptography Conference, TCC 2013, Tokyo, Japan, March 3-6, 2013. Proceedings*, 457–476.

Kearnes, S., McCloskey, K., Berndl, M., Pande, V., & Riley, P. (2016). "Molecular graph convolutions: Moving beyond fingerprints." *Journal of computer-aided molecular design*, *30*, 595–608.

Kipf, T. N., & Welling, M. (2016). "Semi-supervised classification with graph convolutional networks." *arXiv preprint arXiv:1609.02907*.

Kleinberg, J. M. (1999). "Authoritative sources in a hyperlinked environment." *Journal of the ACM (JACM)*, *46*5, 604–632.

Li, P., Wang, Y., Wang, H., & Leskovec, J. (n.d.). "Distance encoding: Design provably more powerful neural networks for graph representation learning."

Li, Y., Yu, R., Shahabi, C., & Liu, Y. (2017). "Diffusion convolutional recurrent neural network: Data-driven traffic forecasting." *arXiv preprint arXiv:1707.01926*.

Lohani, D., Belaïd, A., & Belaïd, Y. (n.d.). An invoice reading system using a graph convolutional network. https://doi.org/10.1007/978-3-030-21074-8{\textunderscore}12

Majeed, A., & Rauf, I. (2020). "Graph theory: A comprehensive survey about graph theory applications in computer science and social networks." *Inventions*, *5*1, 10. https://doi.org/10.3390/inventions5010010

McKay, B. D., et al. (1981). "Practical graph isomorphism."

Monti, F., Boscaini, D., Masci, J., Rodola, E., Svoboda, J., & Bronstein, M. M. (2017). "Geometric deep learning on graphs and manifolds using mixture model cnns." *Proceedings of the IEEE conference on computer vision and pattern recognition*, 5115–5124.

Newman, M. E. (2003). "The structure and function of complex networks." *SIAM review*, *45*2, 167–256.

Pascanu, R., Mikolov, T., & Bengio, Y. (2013). "On the difficulty of training recurrent neural networks." *International conference on machine learning*, 1310–1318.

Prasad, D., Gadpal, A., Kapadni, K., Visave, M., & Sultanpure, K. (2020). "Cascadetabnet: An approach for end to end table detection and structure recognition from image-based documents." *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition workshops*, 572–573.

Qasim, S. R., Mahmood, H., & Shafait, F. (n.d.). "Rethinking table recognition using graph neural networks."

Riba, P., Dutta, A., Goldmann, L., Fornés, A., Ramos, O., & Lladós, J. (2019). "Table detection in invoice documents by graph neural networks." *2019 International Conference on Document Analysis and Recognition (ICDAR)*, 122–127.

Rishi Pal Singh. (n.d.). "Application of graph theory in computer science and engineering."

Saha, R., Mondal, A., & Jawahar, C. (2019). "Graphical object detection in document images." *2019 International Conference on Document Analysis and Recognition (ICDAR)*, 51–58.

Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., & Monfardini, G. (2008). "The graph neural network model." *IEEE transactions on neural networks*, *20*1, 61–80.

Schreiber, S., Agne, S., Wolf, I., Dengel, A., & Ahmed, S. (2017). "Deepdesrt: Deep learning for detection and structure recognition of tables in document images." *2017 14th IAPR international conference on document analysis and recognition (ICDAR)*, *1*, 1162–1167.

Shafait, F., & Smith, R. (2010). "Table detection in heterogeneous documents." *Proceedings of the 9th IAPR International Workshop on Document Analysis Systems*, 65–72.

Singh, H., & Sharma, R. (2012). "Role of adjacency matrix & adjacency list in graph theory." *International Journal of Computers & Technology*, *3*1, 179–183.

Smith, R. (2007). "An overview of the tesseract ocr engine." *Ninth international conference on document analysis and recognition (ICDAR 2007)*, *2*, 629–633.

Smock, B., Pesala, R., & Abraham, R. (2022). "Pubtables-1m: Towards comprehensive table extraction from unstructured documents." *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 4634–4642.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). "Dropout: A simple way to prevent neural networks from overfitting." *The journal of machine learning research*, *15*1, 1929–1958.

Veličković, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., & Bengio, Y. (2017). "Graph attention networks." *arXiv preprint arXiv:1710.10903*.

Wang, X., & Zhang, M. (2022). "Glass: Gnn with labeling tricks for subgraph representation learning." *International Conference on Learning Representations*.

Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., & Philip, S. Y. (2020). "A comprehensive survey on graph neural networks." *IEEE transactions on neural networks and learning systems*, *32*1, 4–24.

Yao, L., Mao, C., & Luo, Y. (2019). "Graph convolutional networks for text classification." *Proceedings of the AAAI conference on artificial intelligence*, *33*01, 7370–7377.

Ying, R., He, R., Chen, K., Eksombatchai, P., Hamilton, W. L., & Leskovec, J. (2018). "Graph convolutional neural networks for web-scale recommender systems." *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, 974–983.

Zhang, M., & Chen, Y. (n.d.). "Link prediction based on graph neural networks."

Zhang, M., Li, P., Xia, Y., Wang, K., & Jin, L. (2021). "Labeling trick: A theory of using graph neural networks for multi-node representation learning." *Advances in Neural Information Processing Systems*, *34*, 9061–9073.

Table 1: Parameter ranges of tuning

K	**BS**[1]	**Layers**	**Dropout**	**HN**[2]	**LR**[3]	**WD**[4]
2	16	2	0.3	256	$1e^{-4}$	0.01
3	32	3	0.5	512	$1e^{-5}$	0.001
4	64	4	0.7	1024	$1e^{-6}$	

[1] Batch Size;

[2] Hidden Neurons;

[3] Learning Rate;

[4] Weight Decay

Table 2: Tuning Results, Link Prediction

K	**BS**[1]	**Layers**	**Dropout**	**HN**[2]	**LR**[3]	**WD**[4]	**Acc.**[5]	**Mac. F1**[6]	**Mic. F1**[7]
2	32	3	0.3	1024	0.00001	0.001	0,9038	0,8916	0,9038
2	16	2	0.7	256	0.0001	0.001	0,9034	0,8912	0,9034
4	16	2	0.7	1024	0.00001	0.001	0,9034	0,8909	0,9034
3	16	3	0.5	512	0.0001	0.001	0,9033	0,8910	0,9033
2	32	3	0.5	1024	0.00001	0.01	0,9031	0,8907	0,9031
3	32	4	0.3	256	0.00001	0.01	0,9031	0,8904	0,9031
2	32	4	0.3	256	0.00001	0.001	0,9030	0,8905	0,9030
2	16	2	0.3	256	0.0001	0.01	0,9025	0,8902	0,9025
2	16	2	0.7	1024	0.0001	0.001	0,9024	0,8903	0,9024
3	32	4	0.3	512	0.0001	0.001	0,9020	0,8899	0,9020
3	64	2	0.7	512	0.0001	0.01	0,9016	0,8884	0,9016
4	16	3	0.5	256	0.00001	0.001	0,9016	0,8885	0,9016

continued

Table 2: Tuning Results, Link Prediction

K	BS[1]	Layers	Dropout	HN[2]	LR[3]	WD[4]	Acc.[5]	Mac. F1[6]	Mic. F1[7]
3	64	4	0.5	256	0.00001	0.001	0,9014	0,8882	0,9014
3	32	3	0.7	256	0.00001	0.01	0,8995	0,8867	0,8995
2	64	3	0.3	1024	0.000001	0.001	0,8980	0,8846	0,8980
3	32	3	0.3	1024	0.000001	0.001	0,8971	0,8838	0,8971
2	16	2	0.3	512	0.000001	0.001	0,8955	0,8822	0,8955
4	32	2	0.3	1024	0.000001	0.001	0,8945	0,8806	0,8945
4	64	3	0.5	1024	0.0001	0.01	0,8939	0,8791	0,8939
2	16	3	0.5	1024	0.000001	0.001	0,8923	0,8787	0,8923
2	64	3	0.3	512	0.000001	0.01	0,8901	0,8740	0,8901
4	32	4	0.3	1024	0.000001	0.001	0,8585	0,8330	0,8585
3	64	3	0.5	512	0.000001	0.01	0,7591	0,6889	0,7591
4	32	4	0.3	256	0.000001	0.01	0,7117	0,6028	0,7117
3	64	4	0.5	256	0.000001	0.01	0,6369	0,3911	0,6369
3	32	4	0.7	1024	0.000001	0.001	0,6213	0,4250	0,6213

[1] Batch Size;

[2] Hidden Neurons;

[3] Learning Rate;

[4] Weight Decay;

[5] Validation Accuracy;

[6] Validation Macro F1;

[7] Validation Micro F1

Performance results are rounded to four decimal places

Table 3: Tuning Results, Straight-odd Prediction

K	**BS**[1]	**Layers**	**Dropout**	**HN**[2]	**LR**[3]	**WD**[4]	**Acc.**[5]	**Mac. F1**[6]	**Mic. F1**[7]
4	32	3	0.3	512	0.0001	0.001	0,7988	0,7986	0,7988
3	16	3	0.3	512	0.00001	0.001	0,7947	0,7947	0,7947
3	16	4	0.5	256	0.00001	0.001	0,7936	0,7933	0,7936
3	16	4	0.7	256	0.0001	0.001	0,7934	0,7933	0,7934
3	16	3	0.7	1024	0.00001	0.001	0,7920	0,7920	0,7920
4	32	2	0.7	1024	0.00001	0.001	0,7916	0,7915	0,7916
3	16	4	0.5	256	0.0001	0.01	0,7886	0,7878	0,7886
2	16	2	0.5	256	0.0001	0.001	0,7878	0,7877	0,7878
3	16	4	0.3	1024	0.00001	0.01	0,7863	0,7862	0,7863
4	16	4	0.5	1024	0.00001	0.01	0,7854	0,7819	0,7854
2	64	3	0.7	512	0.00001	0.001	0,7830	0,7830	0,7830
2	16	3	0.5	256	0.000001	0.01	0,7826	0,7824	0,7826
3	64	2	0.3	512	0.000001	0.01	0,7825	0,7823	0,7825
2	32	2	0.3	256	0.000001	0.01	0,7823	0,7822	0,7823
3	32	3	0.3	256	0.000001	0.01	0,7821	0,7819	0,7821
2	32	4	0.7	1024	0.000001	0.001	0,7817	0,7814	0,7817
2	64	2	0.3	512	0.000001	0.01	0,7806	0,7804	0,7806
2	64	2	0.7	512	0.000001	0.001	0,7796	0,7796	0,7796
2	16	3	0.3	512	0.000001	0.001	0,7792	0,7783	0,7792
4	16	2	0.7	512	0.000001	0.01	0,7792	0,7792	0,7792
2	64	4	0.7	512	0.00001	0.01	0,7788	0,7783	0,7788
2	16	3	0.7	1024	0.00001	0.01	0,7745	0,7734	0,7745
2	16	2	0.7	256	0.00001	0.01	0,7726	0,7725	0,7726
2	16	3	0.7	512	0.0001	0.01	0,7607	0,7577	0,7607

continued

Table 3: Tuning Results, Straight-odd Prediction

K	BS[1]	Layers	Dropout	HN[2]	LR[3]	WD[4]	Acc.[5]	Mac. F1[6]	Mic. F1[7]
4	16	4	0.7	512	0.000001	0.01	0,7595	0,7595	0,7595
2	64	2	0.7	256	0.000001	0.01	0,5379	0,3835	0,5379
4	64	2	0.7	512	0.000001	0.001	0,5285	0,3468	0,5285
3	16	3	0.7	512	0.000001	0.001	0,5283	0,3459	0,5283

[1] Batch Size;

[2] Hidden Neurons;

[3] Learning Rate;

[4] Weight Decay;

[5] Validation Accuracy;

[6] Validation Macro F1;

[7] Validation Micro F1

Performance results are rounded to four decimal places

Table 4: Performance, Straight-odd Prediction

Data Used	Accuracy	Micro			Macro		
		F1	Precision	Recall	F1	Precision	Recall
All difficulties	0,7801 $\pm 0,0162$	0,7801 $\pm 0,0162$	0,7801 $\pm 0,0162$	0,7801 $\pm 0,0162$	0,7746 $\pm 0,0255$	0,7983 $\pm 0,0120$	0,7776 $\pm 0,0242$
Easy graphs	0,8424 $\pm 0,0034$	0,8424 $\pm 0,0034$	0,8424 $\pm 0,0034$	0,8424 $\pm 0,0034$	0,8407 $\pm 0,0062$	0,8515 $\pm 0,0103$	0,8413 $\pm 0,0078$
Medium graphs	0,7592 $\pm 0,0233$	0,7592 $\pm 0,0233$	0,7592 $\pm 0,0233$	0,7592 $\pm 0,0233$	0,7508 $\pm 0,0378$	0,7824 $\pm 0,0128$	0,7559 $\pm 0,0341$
Hard graphs	0,7021 $\pm 0,0389$	0,7021 $\pm 0,0389$	0,7021 $\pm 0,0389$	0,7021 $\pm 0,0389$	0,6853 $\pm 0,0679$	0,7300 $\pm 0,0045$	0,6972 $\pm 0,0527$

continued

Table 4: Performance, Straight-odd Prediction

Data Used	Accuracy	Micro			Macro		
		F1	Precision	Recall	F1	Precision	Recall
All difficulties incl. SLI[1]	0,7825	0,7825	0,7825	0,7825	0,7760	0,7991	0,7788
	$\pm 0,0137$	$\pm 0,0137$	$\pm 0,0137$	$\pm 0,0137$	$\pm 0,0247$	$\pm 0,0124$	$\pm 0,0265$

[1] Single Line Items

$\pm$ indicates the standard deviation for averaged values across three seeds

All values are rounded to four decimal places

Table 5: Performance, Link Prediction

Data Used	Accuracy	Micro			Macro		
		F1	Precision	Recall	F1	Precision	Recall
All difficulties	0,9001	0,9001	0,9001	0,9001	0,8863	0,9007	0,8759
	$\pm 0,0004$	$\pm 0,0004$	$\pm 0,0004$	$\pm 0,0004$	$\pm 0,0005$	$\pm 0,0006$	$\pm 0,0008$
Easy graphs	0,9227	0,9227	0,9227	0,9227	0,9193	0,9286	0,9135
	$\pm 0,0004$	$\pm 0,0004$	$\pm 0,0004$	$\pm 0,0004$	$\pm 0,0005$	$\pm 0,0002$	$\pm 0,0006$
Medium graphs	0,8842	0,8842	0,8842	0,8842	0,8636	0,8795	0,852
	$\pm 0,0004$	$\pm 0,0004$	$\pm 0,0004$	$\pm 0,0004$	$\pm 0,0005$	$\pm 0,0006$	$\pm 0,0008$
Hard graphs	0,8914	0,8914	0,8914	0,8914	0,8458	0,8569	0,8362
	$\pm 0,0007$	$\pm 0,0007$	$\pm 0,0007$	$\pm 0,0007$	$\pm 0,001$	$\pm 0,0012$	$\pm 0,0013$
All difficulties incl. SLI[1]	0,9049	0,9049	0,9049	0,9049	0,8913	0,9042	0,8817
	$\pm 0,0017$	$\pm 0,0017$	$\pm 0,0017$	$\pm 0,0017$	$\pm 0,0021$	$\pm 0,0012$	$\pm 0,0026$

[1] Single Line Items

$\pm$ indicates the standard deviation for averaged values across three seeds

All values are rounded to four decimal places

Table 6: Performance, Line-item prediction

Data Used	Accuracy	Micro			Macro		
		F1	Precision	Recall	F1	Precision	Recall
All difficulties	0,7382	0,7382	0,7382	0,7382	0,7407	0,7336	0,7756
	$\pm 0,0074$	$\pm 0,0074$	$\pm 0,0074$	$\pm 0,0074$	$\pm 0,0537$	$\pm 0,0581$	$\pm 0,0357$
Easy graphs	0,7940	0,7940	0,7940	0,7940	0,8063	0,8101	0,8237
	$\pm 0,0062$	$\pm 0,0062$	$\pm 0,0062$	$\pm 0,0062$	$\pm 0,0452$	$\pm 0,0647$	$\pm 0,0219$
Medium graphs	0,7286	0,7286	0,7286	0,7286	0,6597	0,6468	0,7206
	$\pm 0,0033$	$\pm 0,0033$	$\pm 0,0033$	$\pm 0,0033$	$\pm 0,0604$	$\pm 0,0536$	$\pm 0,0786$
Hard graphs	0,6725	0,6725	0,6725	0,6725	0,4165	0,3731	0,5624
	$\pm 0,0091$	$\pm 0,0091$	$\pm 0,0091$	$\pm 0,0091$	$\pm 0,1229$	$\pm 0,1075$	$\pm 0,1552$
All difficulties incl. SLI[1]	0,7349	0,7349	0,7349	0,7349	0,7755	0,7824	0,7908
	$\pm 0,0026$	$\pm 0,0026$	$\pm 0,0026$	$\pm 0,0026$	$\pm 0,0455$	$\pm 0,0450$	$\pm 0,0383$

[1] Single Line Items

$\pm$ indicates the standard deviation for averaged values across three seeds

All values are rounded to four decimal places

Table 7: Results per target, Straight-odd Prediction

Target label	Precision	Recall	F1
Straight line item	0,7889	0,8203	0,7964
	$\pm 0,0770$	$\pm 0,1215$	$\pm 0,0123$
Odd line item	0,8076	0,7348	0,7529
	$\pm 0,1009$	$\pm 0,1695$	$\pm 0,0623$

$\pm$ indicates the standard deviation for averaged values across three seeds

All values are rounded to four decimal places

Table 8: Results per target, Link Prediction

Target label	Precision	Recall	F1
Incorrect Edge	0,9063	0,8064	0,8535
	$\pm 0,0019$	$\pm 0,0092$	$\pm 0,0060$
Correct Edge	0,9022	0,9554	0,9280
	$\pm 0,0032$	$\pm 0,0002$	$\pm 0,0017$

$\pm$ indicates the standard deviation for averaged values across three seeds

All values are rounded to four decimal places

Figure 1: Parallel Coordinates Plot, Link Prediction

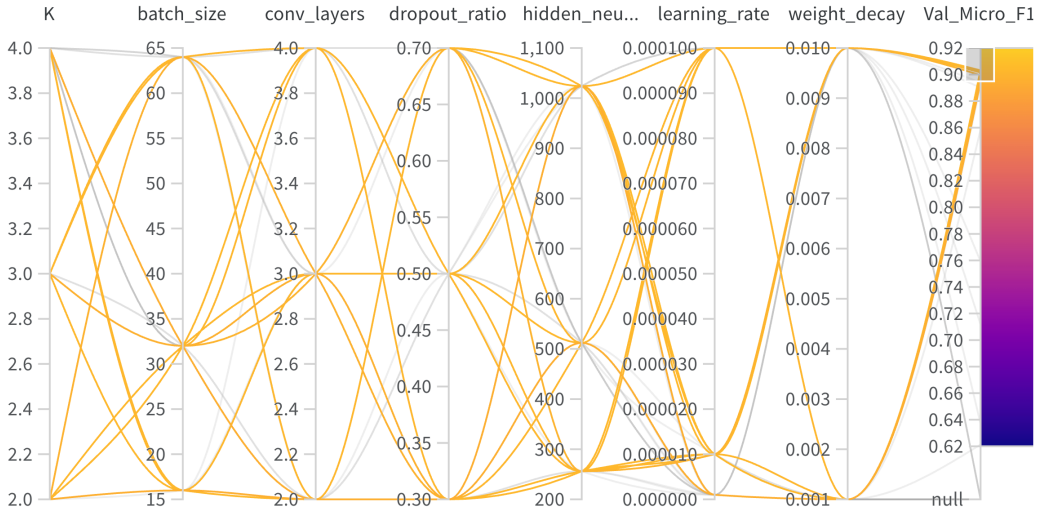

Figure 2: Parameter Importance, Link Prediction

Figure 3: Parallel Coordinates Plot, Straight-odd Prediction

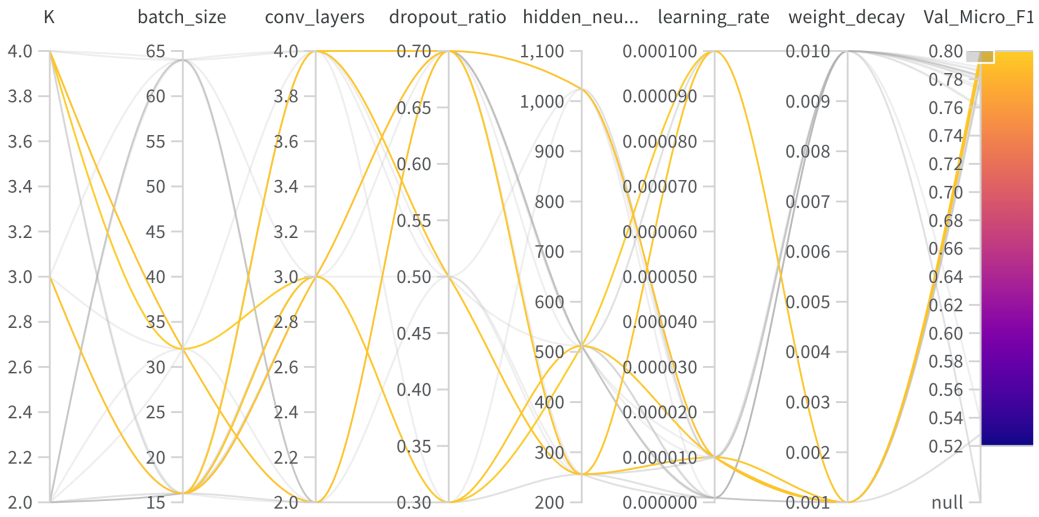

Figure 4: Parameter Importance, Straight-odd Prediction