Using Graph Neural Networks to classify Distribution Graphs from Twitter

by Ferdinand Schaal

A thesis presented for the degree of
Master of science in Human-Centered Artificial Intelligence at
Technical University of Denmark

Technical University of Denmark
Simula Research Laboratory

October 2020

Abstract

Along with the Covid-19 pandemic came a new wave of misinformation that quickly swamped the internet causing serious harm. The spread of misinformation, also known as fake news, has been a frequent problem in the past decade along with the development and the global use of social media. Fake news can cause serious damage such as public panic, have economical consequences and a negative impact on individuals or groups of people. In the light of the misinformation related to Covid-19, several 5G cellular towers were burned down and death threats were made towards individuals working with 5G technology [1].

With the large amount of data that is continuously posted and shared on social media, it is impossible to have a manual oversight. It is therefore necessary to have an automated system that can distinguish fake news from real news. Most previous research in automatic fake news detection has focused on analyzing the content of a text with Natural Language Processing (NLP). Analyzing the content poses a challenging problem in various scenarios when the texts are short, does not explicitly state a false statement or when it contains satirical statements. Furthermore, NLP is also limited to a specific language domain. Additionally to content based methods, there are also methods that look at the propagation of news. These methods typically rely on feature extraction of the propagation trees before applying Machine Learning (ML) methods.

We are proposing a method that uses Graph Neural Networks (GNNs) to classify distribution graphs from the social media platform Twitter. GNNs is a relatively new field of research that makes it possible to directly apply deep learning methods to graph structured data. The distribution graphs are similar to propagation trees, and describe how a tweet is distributed or spread across the platform.

We will work with three different datasets where the first two contains tweets from various German politicians and the third dataset contain tweets that deal with misinformation related to the Covid-19 pandemic. We will compare state-of-the-art GNNs to find the model architecture that results in the best performance on our data.

Preface

This thesis was written at Simula Research Laboratory in fulfilment of the requirements for acquiring M.Sc. in Human-Centered Artificial Intelligence at Technical University of Denmark (DTU) during the period February 2020 to October 2020. This thesis was written under the supervision of Johannes Langguth at Simula and Sune Lehmann at DTU.

Acknowledgment

I would like to thank Johannes Langguth at Simula and Sune Lehmann at DTU for the excellent supervision throughout the work of this thesis. I would further like to thank Daniel Schroeder for all the data collection and supervision of the thesis, as well as Konstantin Pogorelov.

Table of Contents

1 Introduction

Fake news has in recent years gained a lot of public attention after political campaigns such as the United States 2016 presidential elections and the Brexit vote in the United Kingdom, and 2020 saw a new wave of fake news sparked by the Covid-19 pandemic. Fake news is the spreading of misinformation often with an agenda and the goal to influence political views or to distract the public from threatening events. Fast spreading misinformation online is called, when harmful, digital wildfires and can have a serious impact on economics as well as damage to reputation etc. The digital wildfires grow very fast which makes it hard to detect manually ahead of time.

Social media is broadly used around the world, and has quickly become one of the main sources of information. Among one of them is *Twitter*. With 145 million daily active users, the social networking service is a popular domain for posting and sharing news content. Twitter allows users to post short messages, called *tweets*, that other users can read, share or comment. Twitter has around 500 million tweets sent each day, making it impossible to manually control the spread of misinformation on the platform. However, by looking at how the tweets spread on Twitter, one might be able to distinguish misinformation from real news. Vosoughi et al. [2] showed that in the case of political news, fake news compared to true news typically spread further and faster. These types of characteristics can be used to identify fake news when analyzing the propagation of news on Twitter.

The propagation of tweets can be represented as distribution graphs. A distribution graph is a subgraph of the Twitter graph and represents a specific tweet, where the central node is the author of the tweet and the rest of the nodes are users that have shared the tweet. The shared tweets are called *retweets*. The edges are *friend/follower* relationships between the users.

The aim of this thesis is to classify distribution graphs from Twitter with the main agenda of identifying fake news. Before moving on to fake news, we will first look at tweets from German politicians and assess the possibility of classifying tweets from political parties based on the structure of the distribution graphs. In the case of German politicians we will work with two datasets; a small one with a small number of graphs and individual sources, and a bigger one with more graphs and individual sources. The two datasets include politicians from the same political parties. Next we will apply the same methods on a dataset containing misinformation tweets related to the Covid-19 pandemic [3].

Graph Neural Networks (GNNs) will be used to classify the distribution graphs. GNNs has in short time gained a lot of attention within the field of Machine Learning (ML). Previously, neural networks have typically been limited to work with data represented in the Euclidean space. Because of graphs complexity and ability to describe relationships and dependencies, a lot of applications within various fields such as chemistry, computer vision, and Natural Language Processing (NLP), represent the data as graphs. Motivated by deep learning architectures such as Convolutional Neural Network (CNN) [4] and Recurrent Neural Network (RNN) [5], new generalizations have been proposed that allow neural networks to directly operate on graph structured data.

Different GNN architectures will be evaluated in order to find out what is best suited for our task. The GNN framework that we will be using is PyTorch Geometric (PyG) [6]. PyG is a GNN framework that is built on top of PyTorch [7]. PyG allows one to train a neural network on geometric data.

1.1 Research Question

The main objective of this thesis is to apply GNNs on distribution graphs from Twitter. The model should perform better than ML methods that rely on feature engineering. This leads to the following research question:

Can GNNs accurately classify distribution graphs from Twitter?

This is supported by the following sub questions regarding the design:

1. Are GNNs able to identify sources based on spreading patterns in the distribution graphs?
2. Are GNNs able to detect misinformation based on spreading patterns in the distribution graphs?
3. What GNN architectures are best suited for the task?
4. How well do GNNs perform compared too more traditional ML methods?

2 Background

This chapter consists of three parts. In the first section we will introduce and provide a general background on concepts of deep learning that is relevant for the methods used in this thesis. The second part gives a short introduction to graph theory. The third and last section gives an overview of graph neural networks (GNNs), where we present a brief history on GNN and describe some of the state-of-the-art GNN architectures that we will apply and use in later chapters.

2.1 Deep Learning

This section will give a short overview of the most important and relevant properties of deep learning. For an exhaustive text on the topic, the reader is referred to the book by Goodfellow et al. [8], from which most of this section is inspired by.

Deep learning is a class of machine learning that has received a lot of attention in recent years much due to its incredible performance in representation learning. They are many variants of neural networks, and in recent years it has been a boom in research towards these variants. Neural networks are loosely inspired by how the human brain works. A neural network is represented by multiple layers of parameterized differentiable functions, essentially mapping a input vector x to a target vector y. The word 'deep' in deep learning comes from the use of multiple layers in the network. This allows the neural network to learn arbitrary complex functions when enough layers of simple computational units are given. A neural network trained model stands out from other machine learning models in that it can learn the representation of data additionally to making predictions. This means that features do not have to be engineered manually. These kinds of models came a long time before the recent boom in deep learning. They received a lot of attention in the beginning of the 1990s for its theoretical foundations but were impractical to implement. It was not before the middle of the 2000s that these models started to get used by researchers. This was much due to higher computational powers in computers and larger available datasets required to achieve statistical generalizability [8].

2.1.1 Feedforward networks

The simplest variant of neural networks are the feedforward neural networks also known as multilayer perceptrons (MLPs). The goal of feedforward networks is to approximate a function $f*$ that maps some input x to a target value y. The mapping is defined as $y = f(x; \theta)$ where θ is the parameters that results in the best approximation of $f*$ based on training data.

The word feedforward comes from the fact that information is passed through the network. The direction of information is always passed forward, meaning that there is no feedback connection. If there were feedback connections throughout the network, it would result in an extension of the feedforward model called recurrent neural network (RNN) [5]. Another extension to the feedforward model is the popular convolutional neural network (CNN) [4], which is a powerful network that is used a lot in the field of image analysis and computer vision. In other words, feedforward networks play an important role in deep learning and usually form the basis for a lot of important commercial applications.

Feedforward networks are typically chained together with functions forming a directed acyclic graph (dag), hence the word network. The number of functions is the depth of the network, where each function is a layer in the network. The internal layers are called hidden layers, while the last layer is called the output layer. The training data only tells us what the output of

the output layer should be, therefore the network must decide for itself how it should use the internal layers, hence the expression hidden layers. The dimensionality of a layer is the width of the model. We can think of each element, or unit, in a layer as playing a role analogous to a neuron. The analogy comes from the fact that the units receive several inputs and outputs a single scalar value, an activation value based on the inputs. Another way to put it is that the neurons are acting in parallel where each unit represents a vector-to-scalar function.

Linear models are limited by the fact that they can only model linear relations between data and that feature engineering has to be done manually if one wants to model the interaction between the input variables. These limitations do not apply to neural networks since they are defined by multiple affine functions stacked together with non-linear activation functions in between them.

The simplest variant of the multilayer perceptron is the one-layer feedforward network. It is called one layer because it has one hidden layer between the input and output layer and is defined as follows:

$$\boldsymbol{h} = \sigma(\boldsymbol{W_1 x} + \boldsymbol{b_1}) \tag{1}$$

$$\boldsymbol{y} = \boldsymbol{W_2 h} + \boldsymbol{b_2} \tag{2}$$

where σ is the activation of the hidden layer, $\boldsymbol{W}$ are the weights and $\boldsymbol{b}$ are the biases. Each layer has its own weights matrices and biases. No activation is needed for the output layer if we are predicting numerical values, e.g in regression. However, if we are dealing with a classification problem, the output is normally transformed with a softmax function:

$$\boldsymbol{y} = softmax(\boldsymbol{W_2 h} + \boldsymbol{b_2}) \tag{3}$$

The softmax is a multi-class extension of the sigmoid function. The sigmoid function maps a number from $R \rightarrow [0, 1]$. In other words, the sigmoid function is typically used in binary classification problems while softmax works for both binary and multi class problems.

2.1.2 Activations

The activation functions are what makes the neural network non-linear. Without it, the neural network would be nothing but linear function stacked together, which ultimately is just another linear function. They are the real expressiveness of the network.

There exists a lot of activation functions, therefore we will limit us to introduce just a few popular ones. Up to this point two activation functions have been mentioned, the sigmoid and the softmax:

$$sigmoid(\mathbf{x}) = \frac{1}{1 + \exp -\mathbf{x}} \tag{4}$$

$$softmax(\mathbf{x})_i = \frac{\exp \mathbf{x}_i}{\sum_{j=1}^{K} \exp \mathbf{x}_j} \tag{5}$$

Since they both map an input from 0,1, they can be interpreted as probability distributions and are therefore suited for classification problems.

Another popular activation function is the rectified linear unit (RELu). RELu outputs the input directly if it is positive, otherwise, it will output zero. RELu and all the generalizations of RELu are based upon models being easier to optimize if their behavior is closer to linear.

2.1.3 Loss function

The loss function is what we want to minimize when training the network. It represents the "loss" associated with how well the network is predicting the true values. Neural networks are commonly trained using maximum likelihood, which means updating weights in such a way that maximizes the likelihood of predicting the training data. This is done using the negative log-likelihood as the loss function. We maximize the probability of predicting correctly by minimizing the loss function. In classification, this is combined with a softmax layer, which outputs probabilities for the input to belong to the different classes. The minimum of a function is solved by taking the derivative of the function and setting it equal to zero. The logarithm has mathematical properties that makes it practical to use when minimizing the loss. Specifically, small values are difficult to handle numerically, taking the negative logarithm of a small value converts it to large negative value. Furthermore, the derivative of a product is much easier to solve when taking the log of the product first. This comes in handy when calculating the derivative of the likelihood which is a product of probabilities.

Minimizing the negative log-likelihood is the same as minimizing the cross-entropy between the classifiers output distribution $\hat{y}$ and target distribution y. For classification, the categorical cross entropy loss is used, which measures the difference between the output distribution $\hat{y}$ and the target distribution y:

$$\mathcal{L}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = -\sum_{i}^{n} \boldsymbol{y}_i \log\left(\hat{\boldsymbol{y}}_{\boldsymbol{i}}\right) \tag{6}$$

2.1.4 Back-propagation

The procedure of information being sent forward through the network from the input layer to the output layer is called forward-propagation. During training the forward-propagation results in a scalar loss $\mathcal{L}$. After the forward-propagation we want to update the weights in such a way that we are minimizing the loss:

$$\boldsymbol{\theta} = \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\, \mathcal{L}(\boldsymbol{\theta}), \tag{7}$$

This is done by back-propagation. The back-propagation algorithm sends the information from the loss backwards through the network in order to compute the gradient of the loss. Computing the gradient analytically is straightforward, but evaluating the gradient numerically is not a trivial task. Back-propagation uses the chain rule to compute derivatives of composite functions, which is a function that is composed of multiple inner or nested functions. A neural network is essentially a massive nested composite function. Each layer of a feed-forward neural network can be represented as a single function whose inputs are a weight vector and the outputs of the previous layer. The purpose of back propagation is to derive the partial derivative of our class function with respect to each individual weight in the network. It gives us a way of computing the loss for every layer and then relating these losses to the quantity of real interest of partial derivatives with respect to any weight in the network. In other words, back-propagation is essentially applying the chain rule through all the possible paths in the network. Gradient descent is used to update the weights in the network in an iterative manner to minimize the loss function. The weights are updated by taking steps in the negative direction of the gradient.

2.1.5 Optimizers

Using the entire training set to optimize the weights can be computational expensive and sometimes not feasible. Optimization algorithms have been developed to speed up the process.

The most common optimization algorithm is the stochastic gradient descent (SGD). Opposed to gradient descent, SGD uses a single data point instead of the whole training data to do the optimization. Using only a single data point means that SGD usually has a lower convergence rate than gradient descent. SGD is typically used with mini-batches instead of a single data point. Single data points can result in noisy and hereby unstable estimates of the gradient, while mini-batching gives an unbiased estimate of the gradient by averaging the gradient over the mini-batch. The pseudocode for mini-batch SGD can be seen below.

Algorithm 1 Mini-batch Stochastic Gradient Descent

Require Learning rate η
Require Initial parameter $\boldsymbol{\theta}$
while stopping criterion not met **do**
  Sample a mini-batch of m sampoles from the training set $\{\boldsymbol{x^{(1)}}, \ldots, \boldsymbol{x^{(m)}}\}$ with corresponding targets $\boldsymbol{y^{(i)}}$
  Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow +\frac{1}{m}\nabla_\theta \sum_i \mathcal{L}(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
  Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta\hat{\boldsymbol{g}}$
end while

Adam [9], short for adaptive moment estimation, is an extension to the SGD and has been praised for its excellent results. It combines the use of momentum and adaptive learning, which are two popular techniques used in optimization and will be described here.

Even when we are using mini-batching with SGD, our estimates of the derivative will be noisy. This is because we are only working on a subset of the data. The estimate of the gradient might in some cases go in a different direction than the optimal direction. To prevent this, instead of estimating the gradients independently, SGD with momentum uses an exponential average of the gradient to compute the gradient. This gives a closer estimate to the actual gradient. Another reason to use momentum is that it prevents oscillations of the gradients, which typically happens near a local minima in SGD, since the gradient is estimated from an average. This allows for faster learning and convergence.

Adaptive learning is a technique that takes individual learning rates for each parameter at every time step. It is motivated by the fact that the magnitude of the gradients of parameters might vary a lot in size. Finding a single learning rate appropriate for all parameters can be a difficult task. Adam uses the first and the second moment of the gradient to derive the learning rates. The pseudocode for Adam can be seen below.

Algorithm 2 The Adam algorithm

Require: Initial learning rate η (default: 0.001)
Require: Exponential decay rates for moment estimates, ρ_1 and ρ_2 in $[0, 1]$ (default: 0.9 and 0.999)
Require: Small constant δ for numerical stabilization
Require: Initial parameter $\boldsymbol{\theta}$
Initialize 1st and 2nd moment variables $\boldsymbol{s} = \boldsymbol{r} = 0$
Initialize time step $t = 0$
while stopping criterion not met **do**
    Sample a mini-batch of m samples from the training set $\{\boldsymbol{x^{(1)}}, \ldots, \boldsymbol{x^{(m)}}\}$ with corresponding targets $\boldsymbol{y^{(i)}}$
    Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow +\frac{1}{m}\nabla_\theta \sum_i \mathcal{L}(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    $t \leftarrow t + 1$
    Update biased first moment estimate: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\boldsymbol{g}$
    Update biased second moment estimate: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2)\boldsymbol{g} \odot \boldsymbol{g}$ ($\odot$ = element-wise multiplication)
    Correct bias in 1st moment: $\hat{\boldsymbol{s}} \leftarrow \frac{s}{1-\rho_1^t}$
    Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\eta \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}}+\delta}$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
end while

Learning Rate Decay. During the training process of a neural network, it can be beneficial to decrease the learning rate over time. This is known as learning rate decay and is typically done with pre-defined learning rate schedules. Learning rate decay can help reduce the loss in the later stages of the training process, since a higher learning rate might diverge after the loss has decreased to a certain point. Adam computes individual adaptive learning rates for each model-parameter, which means that when using Adam, learning rate optimization is already included in the optimizer. This is known as time-based learning decay. Nevertheless, combining this with another learning decay method can potentially speed up the learning process even more. Step decay is another type of learning rate decay that drops the learning rate by a factor after it has trained on all the training instances (also known as epoch) a certain number of times. Using Adam combined with step decay allows for higher initial learning rates, which again result in faster convergence.

2.1.6 Regularization

Model generalization is a term used to describe a model's ability to perform well on unseen data. Regularization methods are used to reduce the generalization error by constraining a model's coefficients often at a small cost of overall accuracy.

Dropout A highly effective and computationally inexpensive method for regularization is dropout. Dropout weakens the training of the neural network by stochastically removing hidden units in the network, thereby creating a sub-network. For each hidden unit in each hidden layer, the unit will be independently set to zero after it has been computed with a predetermined probability. These sub-networks are created for every mini-batch and gradient update. The probability k of dropping a unit typically ranges from 0.2 to 0.5 One prominent reason to use dropout is to prevent co-adaptation . Co-adaptation in the field of neural networks means that the neurons are highly dependent on each other. Co-adaptation can lead to overfitting in the case where dependent neurons are being affected by the independent neurons receiving "bad" inputs. With dropout, the hidden units cannot co-adapt since all hidden units will not

be present in the same layer.

2.2 Graph theory

Graph theory is the study of properties and applications of graphs. Graphs are mathematical structures that model pairwise relations between objects. The objects of a graph is a set of nodes also known as vertices, and the relation between the nodes is represented by what we call edges or links. To be consistent with the notation we will denote them as nodes and edges respectively throughout this thesis. In this section we will give a brief introduction to graph theory.

2.2.1 Graph types and definitions

There exist many types of graph representations and it is important to distinguish them. The simplest graph representation is the simple graph and is defined as a pair $G = (V, E)$ where V is a set of nodes, and E is a set of two-sets (finite set with two elements) of edges of the graph G. Simple graphs are undirected, meaning that the edges have no orientation, e.g. the edge $\{u, v\}$ is identical to the edge $\{v, u\}$. On the contrary, a directed graph is a graph in which the edges have orientations, e.g. the edge (u, v) is directed from node u to node v. Simple graphs are unweighted, meaning that all edges are equally important. There also exists weighted graphs where each edge has an arbitrary value that represents a weight. The weights are magnitudes that explain the importance of relationships between nodes.

A tree is a connected undirected graph. A tree is a connected graph with N nodes and $N - 1$ edges. A rooted tree is a tree with a designated root node where every edge either points away from or towards the root node. When edges point away from the root the graph is called an *arborescence* (out-tree) and *anti-arborescence* (in-tree) otherwise. Directed Acyclic Graphs (DAG) are directed graphs with no cycles. Cycles are nodes that are connected in a closed chain. These graphs play an important role in representing structures with dependencies.

A complete graph is a graph with unique edges between every pair of nodes. A complete graph with n vertices is denoted as the graph K_n.

2.2.2 Graph measurements and properties

There are a lot of different graphs measurements and properties, which describes some of the characteristics of a graph. We will describe some of them here.

The *shortest path problem* is the problem of finding a path between two nodes such that the sum of the weights of its constituent edges is minimized. For unweighted graphs this is simply the path with the fewest edges. The distance between nodes (i, j) is the number of edges in the shortest path connecting them. The *eccentricity* of a node v is the greatest distance between the node to all other nodes and is denoted ad $e(V) = \max_u d(v, u)$. The diameter d of a graph is the maximum eccentricity of any node in the graphs.

The *clustering coefficient* is a measure of the degree to which the neighbors of a node are linked to each other. Where the degree of a node i is denoted as k_i and represents the number of edges that are connected to i. The local clustering coefficient for node i with degree k_i is defined as:

$$C_i = \frac{2L_i}{k_i\,(k_i - 1)} \tag{8}$$

The clustering coefficient for a graph G is the average local clustering coefficient for all nodes N in G

2.2.3 Graph representation

There are several ways to represent a graph. The representation of the graphs often depends on what one wishes to apply the graphs to.

An adjacency matrix A is a very simple way to represent a graph. The cell $A(i, j)$ represents the edge weight between node i and node j, where the edge weight is 1 for unweighted graphs. It is space efficient for representing dense graphs (graphs with a lot of edges). However, for sparse graphs (graphs with few edges in comparison to the number of nodes), an adjacency list is more efficient. An adjacency list is a way to represent graphs from nodes to lists of edges. It is efficient for iterating over all edges.

For a graph $G = (V, E)$ with n nodes, its degree matrix is a diagonal matrix, $D \in R^{n\times n}$, where

$$D_{ii} = d(v_i). \tag{9}$$

The Laplacian matrix $L \in R^{n\times n}$ of a simple graph $G = (V, E)$ can be defined as

$$L = D - A \tag{10}$$

2.2.4 Graph Isomorphism

Two graphs are known to be isomorphic if they are topologically equivalent, meaning that they have the same structure. If two graphs are isomorphic, then they can only differ by a permutation of their nodes. That means that for isomorphic graphs there exists a mapping between the nodes of the graphs that preserve node adjacencies.

The problem of determining whether two graphs are isomorphic is not a trivial task. There still has not been proven to be a polynomial time algorithm to solve this and it is not known if the problem is NP-Complete. Weisfeiler and Lehman [10] proposed an algorithm, known as the Weisfeiler-Lehman Isomorphism Test also known as the WL Test, to solve this problem heuristically. The WL Test produces what is called a canonical form for each graph. A canonical form of a graph G is a graph G' that is isomorphic to G and represents the whole isomorphism class of G. If the canonical forms of two graphs are unequal, then the graphs are not isomorphic. The WL - test does not provide a certain answer if the canonical forms are equal, meaning that two non-isomorphic graphs can have an equal canonical form. Nevertheless, it is a powerful tool to examine graph isomorphism.

2.3 Graph Neural Network

This chapter describes the foundation and theory behind GNNs. It gives an overview of the different types and architectures of GNNs, with a specific focus on spatial-based Convolutional Graph Neural Networks (ConvGNNs).

2.3.1 History

Neural networks typically work with a fixed number of input units and Euclidean data. GNN is a neural network that operates on graph structured data. Unlike Euclidean data, graph structured data consists of complex relationships and interdependency between objects. Graphs can be highly irregular and can be of arbitrary size i.e. having different numbers of nodes and each node having different numbers of neighbors. This makes it a challenging task for traditional Machine Learning (ML) algorithms to model the data, e.g. in convolution where the number of neighborhood values are assumed to be fixed and organized in a grid. An assumption in ML algorithms is that instances are independent of each other, which does not hold for graphs, since nodes are related to each other with edges. Traditional ML methods used to deal with graph structured data by mapping the data into a simple representation that could be used by the ML algorithms, e.g. vectors of reals. This naive approach resulted in neglecting important topological dependencies between nodes as well as having a final result that may be unpredictably dependent on the preprocessing step.

Sperduti and Starita [11] were one of the first to propose a method to model graph data with neural networks. Besides it's limitations, it was an early motivation for further study of GNNs. Gori et al. [12] were the first to use the term GNN and Scarselli et al. [13] further improved the model. These early studies lies in the category of recurrent graph neural networks (RecGNNs). In the more recent years, convolutional graph neural networks (ConvGNNs) have been the main focus in the field of GNNs most due to its efficiency and convenience to composite with other neural networks. Besides RecGNNs and ConvGNNs, there are other types of architectures such as Graph Autoencoders.

2.3.2 Categories and Analytical Tasks

There are many different ways of categorising the different architectures in GNN. Wu et al. [14] categorizes GNN into four different categories:

- Recurrent Graph Neural Networks (RecGNNs)
- Convolutional Graph Neural Networks (ConvGNNs)
- Graph Autoencoders (GAEs)
- Spatial-temporal Graph Neural Networks (STGNNs)

We will only focus on RecGNNs and ConvGNNs. For an overview of GAEs and STGNNs, the reader is referred to Wu et al. [14].

GNNs can perform various types of analytical tasks such as node prediction, link prediction and graph prediction. Node prediction aims at predicting node labels, link prediction aims at predicting future edges in dynamic graphs and graph prediction aims at predicting labels for an entire graph. The best suited GNN architecture can vary based on the particular analytical task of interest. However, they are also closely related since the only difference between node and graph prediction is that a pooling layer has to be applied on the node representation to do graph prediction, and the classification or regression has to be independent from the properties of each node (this is described in more detail in section 2.3.4).

Our objective is to do classification on graphs, and we will therefore only work with graph prediction.

2.3.3 Recurrent Graph Neural Networks

The first types of neural network architectures modelling graphs fall under the category of RecGNNs. Sperduti and Starita [11] claimed that feature based modeling is sensitive to the features selected for the representations and are unable to represent any specific information about the relationships among components. Sperduti and Starita proposed a generalization of the standard neuron; extending neural networks to process structures. They adapted existing learning algorithms to represent and classify structured patterns. Specifically, they applied neural networks to directed acyclic graphs (DAG), and were able to label graphs of different sizes and complexity. Standard neurons can only process instructed patterns while recurrent neurons can only process sequences. Sperduti and Starita's generalized recursive neuron is an extension of the traditional recursive neuron. Additionally, in using the output of the unit from the previous time step, Sperduti and Starita considers the outputs of the unit for all the nodes which are pointed by the current input node.
Sperduti and Starita's work was an early motivation for further study of GNNs. Nevertheless, it suffered from several limitations. It only processed DAGs and was limited to graph prediction.

Gori et al. [12] proposed a method that worked on both graph and node prediction. Scarselli et al. [13] further improved this method and presented a model that is often considered the fundamental architecture of GNN. Liu and Zhou [15] refers to Scarselli et al. [13] as the Vanilla Graph Neural Network. We will give a detailed description on the vanilla GNN since it is very much needed in-order to understand the fundamental principles of all GNNs.

The idea emphasized in the vanilla GNN is that the nodes represent objects or concepts and the edges represent the relationship between nodes. Each object is defined by features and related objects. The proposed architecture demonstrated how graphical information can be encoded into a set of *states* $\boldsymbol{x} \in R^2$ associated with a graph's nodes, where the states are updated in an iterable fashion with respect to the topological relationship between the nodes. The encoded states are used to generate an output o_n (i.e. node or graph label distributions). The parametric functions f_w and g_w, respectively called local transition function and local output function, are used to update the state of each node and to produce the output. They are defined in the equation below:

$$x_n = f_w(l_n, l_{co[n]}, x_{ne[n]}, l_{ne[n]}) \tag{11}$$

$$o_n = g_w(x_n, l_n) \tag{12}$$

where l_n, $l_{co[n]}$, $x_{ne[n]}$, $l_{ne[n]}$ are the label of n, the labels of n's edges, the states and the labels of the nodes in the neighborhood of n. Figure 1 gives an example of how the state of a node is computed by the neighborhood information of the node 1.

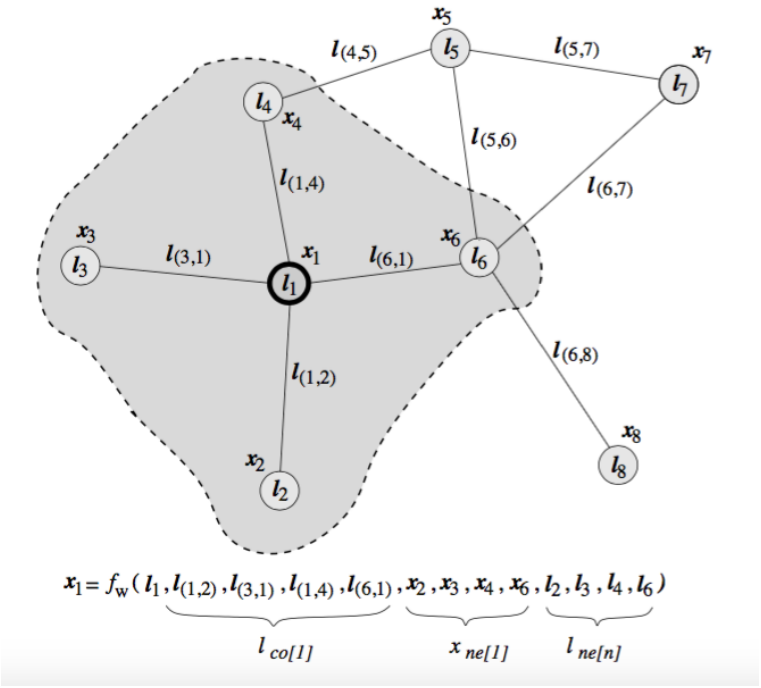

Figure 1: Figure 2 from Scarselli et al. [13], visualizing a simple graph and how the state x_1 for node 1 is calculated by information from it's immediate neighborhood and itself.

By stacking all of the variables, the outputs, the labels, and the node labels, we can re-write the local functions to take the glabal form:

$$x = F_w(x, l) \tag{13}$$

$$o = G_w(x, l_N) \tag{14}$$

F_w is denoted as the global transition function and G_w is denoted as the global output function. The vanilla GNN uses the Banach fixed point theorem [16] to find a unique solution of the system of equations, so that when x and o are unique Eq. (3) and Eq. (4) defines a map $\varphi_w : D \to R^m$ which in any given graph returns an output for each node. As suggested by the Banach's fixed point theorem the state is computed in the following iterative fashion:

$$x(t+1) = F_w(x(t), l) \tag{15}$$

x_t denotes the ith iteration of x. For any initial value of $x(0)$ the system in Eq. 5 converges exponentially fast to the solution of Eq (3).

A loss function is defined to learn the parameters of f_w and g_w and can be written as:

$$loss = \sum_{i=1}^{p} (t_i - o_i), \tag{16}$$

where t_i is the target information for node i and p is the number of supervised nodes. The states x_n^t are iteratively updated by Eq. 1 until time step T. A fixed point solution for Eq 3 is then approximated by $H(T)$. The gradients of the weights W are computed from the loss where W is then updated accordingly to the gradients. The update of W is carried out within the traditional framework of gradient descent.

The vanilla GNN is able to train a model for a specific supervised/semisupervised problem

using various types of graphs (e.g. acyclic, cyclic, directed, un-directed). Besides its limitations, e.g. the inefficient iterative update of hidden states and its shared parameters throughout all layers , it is the first GNN model to properly incorporate neural networks into the graph domain. Following [13] several RecGNNs were proposed to increase training efficiency and to make RecGNNs more scalable.

2.3.4 Convolutional Graph Neural Networks

Motivated by the success of CNNs in computer vision, CNNs where adapted to graph data. ConvGNN generalized convolution on grid data to graph data. Like RecGNNs, ConvGNN represent the state of a node by aggregating a nodes features and its neighbors' features. The big difference between RecGNNs and ConvGNNs is that ConvGNNs uses different weight parameters for each layer, while RecGNNs uses the same weights for each layer. ConvGNNs stack multiple layers to extract high-level node representation. ConvGNNs can be divided into spectral-based approaches and spatial-based approaches.

Spectral graph theory exploits the properties of a graph relationship to characteristic polynomial, eigenvalues and eigenvalues of matrices related to the graph. Spectral models have a theoretical foundation in graph signal processing. They either have to perform eigenvector computation or handle the whole graph at the same time. Spectral models rely on graph Fourier basis and generalize poorly on new graphs. They assume fixed graphs and are limited to undirected graphs.

Spatial models are mostly preferred over spectral models due to efficiency, generality and flexibility. Spatial models scale better to large graphs, since they directly perform convolutions in the graph domain via information propagation. Computation can be done in batch-wise. Graph convolutions are performed locally on each node, weights can easily be shared across different locations. We will only work with spatial-based ConvGNNs and we will therefore not go into detail on how the spectral-based convGNNs work.

Spatial-based ConvGNNs perform convolution of a node's neighborhood to update the states of the nodes. The convolution operation uses the same principle of neighborhood aggregation as the vanilla GNN model described in Section 2.3.3. In ConvGNNs, the process of deriving node states is typically denoted as node embedding.

Node embedding is done in such a way that nodes that are related in the graph are mapped close together in the embedding space. The embeddings are based on local neighborhoods of nodes. Each node in a graph defines a computational graph based on its spatial information. This means that each node has its own neural network that is defined by its computational graph. The neural networks learns how to propagate and aggregate neighborhood information to derive representations of the nodes. The propagation captures both structure information from the graph as well as it learns how to combine the neighborhood node information to enrich the representation of a node. Node embedding is done at each layer of the network. How the computation graphs are defined and how information is being propagated is illustrated in Figure 2 below.

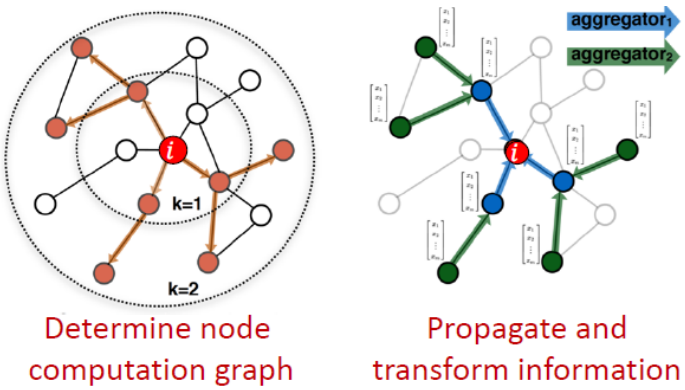

Figure 2: Illustrations from Leskovec [17]. The illustration to the left shows how the computational graph for the target node i is determined by its neighborhood. In this example $k = 2$ which means that i's neighborhood is defined by 2 hops of neighborhood information. In the illustration to the right, information is first aggregated from i's second hop neighborhood to its first hop neighborhood (marked in green), then information is aggregated from the first hop neighborhood to node i marked in blue.

The depth of the network i.e. number of layers, describes how many hops of neighborhood information is going to be propagated. In Figure 2, the computation graph of i is determined based on 2 hops of local neighborhoods. Figure 3 below gives a concrete example of a node embedding and illustrates what the computational graph looks like.

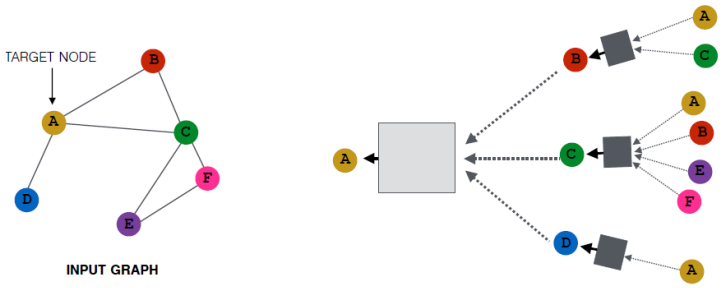

Figure 3: Illustrations from Leskovec [17]. The figure illustrates how the node embedding of target node A in a graph (illustrated on the left) is done. As seen in the illustration on the right, information is initially propagated from the farthest neighbors of node A, where in this example $k = 2$. The information that is propagated in layer 0 (farthest to the right) is the node features of the respective nodes. The nodes are aggregated together in the black boxes in a non-linear aggregation and combined with the node at the current layer. Please note that node A is included in all three aggregations at layer 1. This is because a node is always one of the node's neighbors of the neighbors.

Same as for the example in Figure 2, we have $k = 2$, which means we are taking two hops from the target node A to define its neighborhood. The propagation scheme in Figure 3 is visualized in a tree like structure with the target node A as the tree root. The node embedding at layer 0 are essentially the node features. This is the simplest representation of the node. If the node features are not given or are excluded by choice, one-hot encoding can be used (we will describe the injective importance of this later). The black boxes in Figure 3 illustrate a non-linear transformation that aggregates the node embeddings at the previous layer and combines it with the node at the current layer. The aggregation must be order invariant, meaning that we want to have the same node embedding no matter the permutation of the nodes. The weights in the same layer across each computational graph are shared, but as opposed to the vanilla

GNN each layer in the computational graph has a different set of weights. Note that the nodes at layers above layer 0 are latent representations of the nodes at that current layer. The black boxes are what differentiate the different architectures. This can generally be written as:

$$a_v^{(k)} = \text{AGGREGATE}^{(k)}\left(\left\{h_u^{(k-1)} : u \in \mathcal{N}(v)\right\}\right), h_v^{(k)} = \text{COMBINE}^{(k)}\left(h_v^{(k-1)}, a_v^{(k)}\right), \quad (17)$$

Where $h_v^{(k)}$ is the node embedding at the k'th layer. As mentioned above $h_v^{(0)}$ is initialized to X_v. $\mathcal{N}(v)$ is node v's neighbors, i.e. nodes adjacent to v. The AGGREGATE function collects neighborhood information to extract the aggregated feature vector $a_v^{(k)}$. The COMBINE function combines the aggregated node features with the previous node feature. The modelling choice for the AGGREGATE and COMBINE functions varies based on the particular GNN architecture.

GraphSAGE

Hamilton et al. [18] proposed a general inductive framework, GraphSAGE(SAmple and aggreGatE), to generate node embeddings. They extended the GCN architecture [19] to the task of inductive unsupervised learning approach, which had previously only been applied in a transductive setting with graphs of fixed size. GraphSAGE generalizes the GCN approach with trainable aggregation functions.

GraphSAGE uses a sampling technique to keep the computational footprint of each batch fixed by uniformly sampling a fixed set of neighbors for the aggregation. After the aggregation of neighborhood information, GraphSAGE concatenates the aggregated neighborhood vector with the node's current representation. A fully connected layer with a nonlinear activation function is then applied to the concatenated vector to get the representation at the next time step.
GraphSAGE uses three types of order invariant aggregators.

Mean aggregator. The mean aggregator takes the element wise mean of the vectors in $(h_v^{(k-1)} \cup \{h_u^{(k-1)}, u \in \mathcal{N}(v)\})$. This variant of GraphSAGE is usually called GraphSAGE-GCN since it is inspired by the convolutional propagation in the transductive GCN framework. The mean-based aggregator can therefore be called convolutional, since it is a linear approximation of a localized spectral convolution. The GrapgSAGE-GCN is formulated as:

$$h_v^{(k)} \leftarrow \sigma\left(W \cdot \text{MEAN}\left(h_v^{(k-1)} \cup \left\{h_u^{(k-1)}, u \in \mathcal{N}(v)\right\}\right)\right), \quad (18)$$

where W is a learnable matrix and σ is an activation function. A big difference in GraphSAGE-GCN compared to the other variants of GraphSAGE is that the COMBINE step in equation 12 is not a concatenation but a summation. Hamilton et al. [18] showed that the use of concatenation instead of summation, can lead to significant gains in performance by providing a "skip connection" between the layers.

LSTM aggregator. A more complex aggregator is the LSTM aggregator. The LSTM aggregator is based on the LSTM architecture [20]. It has more expressive power than the mean aggregator. To make the aggregator permutation invariant, random permutation of node's neighbors has to be applied.

Pooling aggregator. The pooling aggregator is both permutation invariant as well as trainable. Instead of transforming the data after the aggregation as described above, the pooling aggregator applies a fully connected layer to each neighborhood vector before aggregating the information using an element wise max pooling.

$$\text{AGGREGATE}_{(k)}^{(pool)} = \max\left(\left\{\sigma\left(W_{pool}h_{u_i}^{(k)}\right), u \in \mathcal{N}(v)\right\}\right), \tag{19}$$

GNN and the WL-test

There is a strong resemblance in how GNN is performing the node embedding and how the WL test is performing the node coloring. For instance, in GraphSAGE, the algorithm can be viewed as a continuous approximation of the WL test where the hash function is replaced by a trainable neural network aggregator. If the aggregation function is injective and the WL algorithm maps two graphs to different colors the GNN will map the two graphs to two different embeddings as well.

Note that GNNs objective is not to test for graph isomorphism but to learn to generate useful node representation. The WL-test is too computationally expensive and restrictive in the sense that graphs must be identical to be assumed similar. However, the WL test provides a theoretical framework for how GNNs are learning topological structure of the graphs.

Graph Isomorphism Network

Xu et al. [21] proposed a new architecture, Graph Isomorphism Network (GIN), that builds upon the limitation of GraphSAGE by allowing arbitrary aggregation functions on multisets. They present a theoretical framework for analyzing the representational power of GNNs with higher discriminative power than previous proposed methods. GIN is inspired by the close connection between GNN and the WL test. They prove that GNNs is at most powerful as the WL test in distinguishing graph structures and reveals that GCN and GraphSAGE can not distinguish certain simple graph structures.

As described and illustrated in Figure 2 and 3, the node embedding of the GNN updates each node's feature vector by capturing the structure and features of its immediate neighbors i.e. its rooted subtree structures. Xu et al. [21] describes the feature vector of a set of neighboring nodes as a multiset; meaning that it allows multiple instances of itself i.e. an element can appear multiple times since different nodes can have identical feature vectors.

Definition 1 (Multiset) (Xu et al. [21]). A multiset is a generalized concept of a set that allows multiple instances for its elements. More formally, a multiset is a 2-tuple $X = (S, m)$ where S is the underlying set of X that is formed from its distinct elements, and $m : S \to N_{\geq 1}$ gives the multiplicity of the elements [21].

They argue that a maximally powerful GNN must map two nodes to the same location if and only if they have identical subtree structures with identical features. Since the subtree structures are described recursively as seen in Figure 3, the question can be reformulated to the mapping of two multisets instead of two nodes.

The GNN must be able to aggregate different multisets into different representations, meaning it has to be injective i.e. preserving the distinctness of inputs. Different aggregation functions will have different ways of representing the multisets. Lemma 2 in Xu et al. [21] states that if a

GNN maps non-isomorphic graphs to different embeddings, the WL-test will also decide that the graphs are non-isomorphic.

Lemma 2 (Xu et al. [21]). *Let G_1 and G_2 be any two non-isomorphic graphs. If a graph neural network $A : G \to R^d$ maps G_1 and G_2 to different embeddings, the Weisfeiler-Lehman graph isomorphism test also decides G_1 and G_2 are not isomorphic.*

They follow up Lemma 2 by proving, in Theorem 3, that there exist GNNs as powerful as the WL-test when the neighborhood aggregation function and graph level readout functions are injective.

Theorem 3 (Xu et al. [21]). *Let $A : G \to R^d$ be a GNN. With a sufficient number of GNN layers, A maps any graphs G_1 and G_2 that the Weisfeiler-Lehman test of isomorphism decides as non-isomorphic, to different embeddings if the following conditions hold:*
a) A aggregates and updates node features iteratively with

$$h_v^{(k)} = \phi\left(h_v^{(k-1)}, f\left(\left\{h_u^{(k-1)} : u \in \mathcal{N}(v)\right\}\right)\right) \tag{20}$$

where the functions f , which operates on multisets, and ϕ are injective.
b) A's graph-level readout, which operates on the multiset of node features $\left\{h_v^{(k)}\right\}$ is injective.

Theorem 3 generalizes the WL-test by learning an injective mapping of the subtrees. By mapping the subtrees to the embedding space, the GNN is capturing the similarity of graph structures. This is an important distinction between GNN and the WL-test. The WL test uses one-hot encodings of the node features vectors and is therefore limited to only learn to discriminate different structures. Unlike the WL test, GNNs can go beyond discrimination and are able to capture dependencies between graph structures. This helps with generalization when noisy edges and node features are present and the co-occurence of subtrees is sparse across different graphs.

With Theorem 3 providing conditions for a maximal powerful GNN, Xu et al. [21] develops a theory of "deep multisets", i.e. parameterizing universal multiset functions with neural networks, to be able to model injective multiset functions for the neighbor aggregation. They state in Lemma 5 that the sum aggregator can represent universal functions over multisets and thus it is injective.

Lemma 5 (Xu et al. [21]). *Assume $\mathcal{X}$ is countable. There exists a function $f : \mathcal{X} \to R^n$ so that $h(X) = \sum_{x \in X} f(x)$ is unique for each multiset $X \in \mathcal{X}$ of bounded size. Moreover, any multiset function g can be decomposed as $g(X) = \phi(\sum_{x \in X} f(x))$ for some function ϕ.*

There is an important difference between deep multisets and sets, and that is that certain injective set functions are not injective multiset functions. To give a better intuition of these differences, a simple illustration is given below where the mean, max, and sum aggregation functions are ranked by their expressive power.

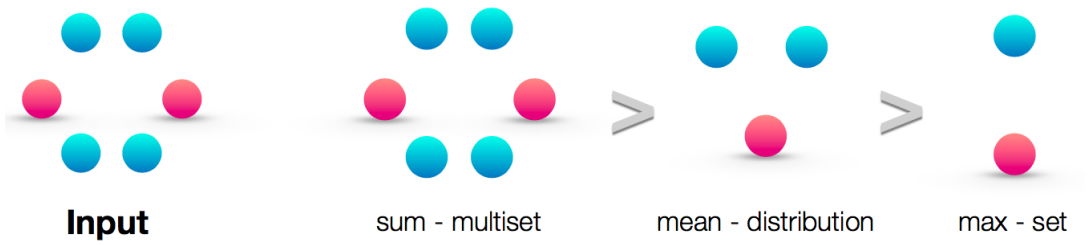

Figure 4: Illustration from Xu et al. [21]. The figure illustrates how the expressive power of the sum, mean and max aggregator are ranked. In this example we have an input of six nodes where the colours represent node features. The sum opperator results in the full mutliset of the nodes, while the mean opperator results in the distribution of the node featuers and the max operator reduces the multiset input to a set.

The max operation converts the input to a set, where you have a single element for each distinct node feature. The mean operation results in a distribution of the node features i.e. the proportion of elements of a given type. The sum operation captures the full multiset. In the next figure, examples are given where mean and max aggregation function fail to distinguish nodes v and v' immediate neighborhood, even though they are different.

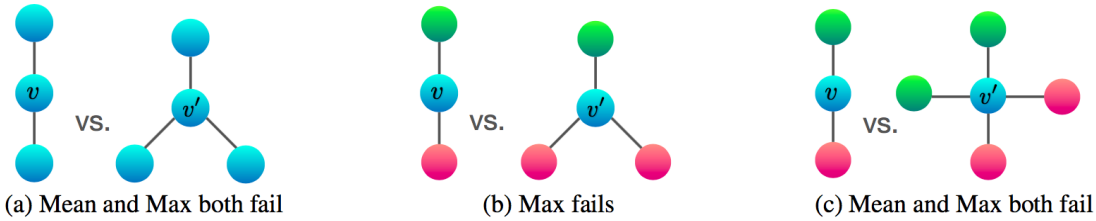

Figure 5: Illustration from Xu et al. [21]. The figure illustrates three examples of the neighborhoods of node v and v', where in each example the neighborhoods differ from each other. In (a) and (c) both mean and max fails to distinguish them, and in (b) max fails.

Figure 5 shows that the aggregation functions used in both GCN and GraphSAGE would not be injective in certain scenarios.

As a result of the universal approximation theorem [22], Xu et al. [21] are able to use MLPs to model and learn both ϕ and f in Lemma 5. Since MLPs can represent a composition of functions, one MLP is used to learn both ϕ and f. Note that if the input features of the first layer are one-hot encodings, an MLP does not have to be applied since the summation of one-hot encodings is injective. With this the node embedding in GIN is defined as:

$$h_v^{(k)} = \text{MLP}^{(k)}\left((1+\epsilon)h_v^{(k-1)} + \sum_{u \in \mathcal{N}(v)} h_u^{(k-1)}\right), \tag{21}$$

where ϵ is a fixed or learnable parameter used to distinguish the root node.

For graph classification tasks, an additional readout function has to be defined. The readout function provides a graph embedding from the embedded node features. And is defined as:

$$h_G = \text{READOUT}\left(\left\{h_v^{(K)} \mid v \in G\right\}\right) \tag{22}$$

Node representations get more refined as the number of iterations (or layers) increases. Deep subtree structures help with achieving high discriminative power, but the representations

at earlier iteration may generalize more than on the later iterations. GIN uses therefore a concatenation (similar to Jumping Knowledge Networks [23]) of the graph representation over all the iterations.

$$h_G = \text{CONCAT}\left(\text{READOUT}\left(\left\{h_v^{(k)} \mid v \in G\right\}\right) \mid k = 0, 1..., K\right) \tag{23}$$

They prove that by replacing the readout with a summation, it generalizes the WL test and the WL subtree kernel.

Other Variants of ConvGNNs

In Addition to GraphSAGE and GIN, we will be using three other GNNs for classification, namely Edge-Conditioned Convolution (ECC) [24], DiffPool [25], and Deep Graph Convolutional Neural Network (DGCNN) [26]. Since they all fall under the category of Spatial-based ConvGNNs, we will only give a brief description that includes some important properties that distinguish them from the other GNNs that have already been mentioned. For a detailed description of ECC, DiffPool, and DGCN, the reader is referred to the respective papers.

ECC uses edge features, such as edge types, instead of node features for node embeddings. The network learns a distinct parameter for each edge feature, and uses the parameters for the neighborhood aggregation in the convolutional layers. Furthermore, ECC uses hierarchical pooling that can be seen as the READOUT operation in Equation 23. Instead of applying a READOUT operation on the final node embeddings to form a graph-level representation as in GIN and GraphSAGE, hierarchical pooling is applied after each convolutional layer in order to reduce the size of the graph and to explore hierarchical structures in the graphs [27]. Specifically, ECC performs hierarchical pooling by eigenvalue decomposition. It splits the graphs into two components by the sign of the largest eigenvector of the graph Laplacian. These pooling methods are independent of the graph convolution, meaning that they are not trained in an end-to-end fashion. DiffPool proposes a differentiable hierarchical clustering method that makes it possible to jointly train it with the graph convolutions.

DGCNN differs from the other GNNs by the use of sortpool layers. The sortpool layer sorts and aligns the node embeddings after the final convolutional layer. The sorted graph representations are then sent through traditional convolutional and dense layers.

3 Related Work

Fake news detection has been a hot topic in research in recent years. There are numerous ways of tackling the challenging problem of fake news. In this chapter, we will give an overview of the methods that has been proposed for fake news detection, as well as how GNNs have been applied in hybrid approaches within the field.

Zhou et al. [28] categorizes the main methods used for fake news detection into four classes:

- Knowledge-Based
- Style-Based
- Propagation-Based
- Source-Based

3.1 Knowledge-Based Fake News Detection

Knowledge-based approaches, also known as fact-checking, investigates the authenticity of news by extracting knowledge and comparing it with known facts. The extraction can be done either manually or automatically.

Manual fact-checking is typically done either by expert-based or crowd-sourced manual fact-checking. Expert-based manual fact-checking is a fact-checking method that is usually done by a group of experts within the domain of interest. This is done in small groups by highly credible individuals. There also exists websites where fact-checking is done on demand by experts in different fields such as American politics (PolitiFact [29], FactCheck [30]), Hollywood and celebrities (GossipCop [31]), and ambiguity (HoaxSlayer [32]). This approach is highly accurate, but can be quite expensive and does not scale with the ever growing information on social media. Crowd-sourced manual fact-checking is done by a large group of regular individuals. This usually has less accurate results than expert-based methods, due to political biases and subjective opinions. There also exists websites for this such as Fiskkit [33].

Computational oriented fact checking uses Natural Language Processing (NLP) and Machine Learning (ML) to do automatic fact checking. Knowledge is extracted in a set of (Subject, Predicate, Object) (SPO) triples that should well-represent the given information. SPOs that are verified as true are seen as facts. A fact is usually represented by a knowledge graph (KG), where entities of the facts are represented as nodes and their relationships are represented as edges. In the process of computational oriented fact checking, KGs are first extracted from the web, which is typically done open-source fusing knowledge from several sources. These KGs are used as ground truth when verifying a text/article. An obvious concern for this approach is the sources from which facts (KGs) are extracted.

3.2 Style-Based Fake News Detection

Similar to knowledge based approaches, style-based methods look at the content of an article. However, instead of evaluating the authenticity of an article, as in knowledge-based methods, style-based methods aims at assessing an article´s intention to mislead the reader. This is done by analyzing the style of the article with the assumption that fake news has a certain style that can be distinguished from real news.

The problem is typically defined as a binary classification problem where a news article is predicted to be either true or fake based on extracted machine learning features. There are numerous types of textual features that can be used such as lexicon or syntax features. Lexicon features assesses the frequency statistics of lexicons. A simple method for this is by Bag of Words (BOW). BOW counts the occurrences of each individual word of a text. With the frequencies of words, patterns can be identified and used to detect fake news [34]. Deep syntax is an approach that utilizes syntax features with Probalistic Context Free Grammar (PCGF) [35] [36][34]. PCGF transforms sentences into rules that are describing the syntax structure. Comparing the syntax structure with known structures of misinformation, can prove whether or not a text is fake or real.

Even though it is less common, style-based methods are not just limited to textual features, they can also rely on image features that are extracted from images in news articles.
For the actual classification of style-based methods, ML methods have been applied such as SVMs [35][36], Random Forests (RF) [34], and XGBoost [37][34]. Deep learning models have also been used on embedded text such as Text-CNN [38].

Zhou et al. [34] show that for political news articles, when compared to true news, fake news has higher informality (swear words), diversity (unique verbs), subjectivity (report verbs), more emotional (emotional words), longer sentences and smaller words (characters per word). They also show that images used in fake news articles (based on data from Twitter and Weibo), compared to images in true news articles, typically have higher clarity and coherence, while lower diversity and clustering scores. These characteristics may vary a lot depending on domains, languages, country etc., and they might also evolve over time, which can potentially reduce the prediction performance [36]. Since style-based methods are only relying on the content, they can tackle fake news at an early stage before it has spread on social media. On the down side, the style of fake news is not permanent. Discoveries as mentioned above can inspire fake news creators to change their style, resulting in a never ending vicious circle.

3.3 Propagation-Based Fake News Detection

Propagation-based methods aim at detecting fake news based on propagation, e.g. how the news spreads. These methods use either news cascades (direct representation of news propagation) or self-defined graphs (indirect representation that captures additional information on news propagation). We will only describe the news cascade methods here.

A news cascade is a tree structure that captures the propagation of a news article on a social network. The user who first shared an article is represented by the root node of the tree. The other nodes of the tree are users who have shared the article from their parent node and are connected by directed edges. News cascades are typically represented either as hop-based news cascades (number of hops the news has propagated) or as time-based news cascades (the number of times an article is shared).

A news article can have several news cascades linked to it when one has multiple initiating users. It is often common to include additional information to represent the nodes, such as comments, user information (age, followers, friends) etc. The news cascades can be classified, after features are extracted from the news cascades, with ML methods such as SVM, decision trees, decision rules, naive bayes [39] and RF [40]. Deep learning models has also been used to classify news cascades such as Ma et al. which developed a RvNN [41].

Research has shown that for political news, fake news typically has greater cascade depth, maximum breadth, size, and average distance among all pairs of nodes compared to true news [2]. Furthermore, they showed that the time it takes to reach any depth of a cascade for fake news is less than for true news.

Obviously, propagation-based method can not do early detection of fake news since it relies on the spreading of the news to do the classification, but it is robust against style manipulation. Because it does not rely on the content, the same methods can be applied on news from different language domains.

3.4 Source-Based Fake News Detection

Source based methods perform fake news detection based on assessment of the credibility of the sources that have either created, published, or shared a news article. Analyzing patterns and differences in unreliable vs reliable authors and publishers can help assess the credibility of unknown authors and publishers. For a detailed description of how source based methods works, the reader is referred to Zhou et al. [28].

3.5 GNN in Fake News Detection

Since GNN is a relatively new field, there is not much research done on fake-news detection with GNN. However, Huang et al. [42] purposes a hybrid model that combines a GNN with a recursive neural network (RvNN) to detect rumor spreading on twitter. The RvNN uses news cascades as input and is based on the proposed method in Ma et al. [43]. The news cascades are propagation trees which consist of a source tweet with text data, and leafs of responses and retweets. Each tree is referred to as a claim. The propagation tree structure encoder intends to capture the structural and textural properties of the propagation trees. The main idea behind the RvNN is that the high-level representation of tree nodes gets strengthen by the recursion following the propagation structure over different branches in the tree. By looking at the textual information, a node can for instance support or discourage a parent node which impacts the credibility of the tweet. Huang et al. [42] uses a bottom-up as well as a top-down method for the RvNN. A bottom-up RvNN goes recursively from the leaves at the bottom to the root at the top. A top-down RvNN has the opposite architecture where the RvNN starts from the root leave.

The GNN is used as an user encoder which encaptures user attributes and behaviors. Specifically, they use a GCN for the user encoder. The GCN is only applied on one large undirected graph $G = (V, E)$. The nodes V are all the users in the data where each node has statistical features such as age, number of followers, number of friends etc as node features. Users that appears in the same claim are linked together with unweighted edges E. For each node in V the GCN computes user embedding v_u.

The third part of the model in Huang et al. [42] is an integrator, which combines the output of the RvNN and GCN to classify if a given claim i is a rumor or not. For each claim, the integrator only uses the user embedding v_{u_i} of the root node.

The proposed method proves to be more efficient than previous methods, but the importance of the GNN in the user encoder is questionable, since the model performs best on average with a top-down RvNN without the user encoder. This proves that the promising results are due to the RvNN and not because of the GNN.

4 Data

We will work with three different datasets that are collected from Twitter. Dataset 1 and Dataset 2 contain distribution graphs from German politicians. Dataset 3 contains distribution graphs that are labeled as fake news and non-fake news. This chapter will give a detailed description of what the distribution graphs are, how it is collected, and how it is handled. Additionally, each dataset will have its separate section where its properties and the necessary preprocessing steps are explained.

4.1 Data Collection

Among social networks, Twitter has become the main platform for research. This is most likely due to the fact that data is public and can be accessed through Twitter's API. However, accessing data from Twitter efficiently on a large scale can be a challenging task.

Schroeder et al. [44] proposed the FACT framework for capturing and analysing Twitter data efficiently. FACT uses parallel and distributed cloud-based computing to be able to capture large amounts of data. The datasets used in this thesis are collected through an application of the FACT scraper. We will give a simplified explanation of how the data is collected. For a detailed description on the FACT framework, the reader is referred to the paper [44].

For Dataset 1 and Dataset 2, tweets belonging to specific German parties are extracted from a large number of tweets that are scraped from Twitter. Graphs are built from these tweets, by finding retweets of the respective tweets collected. The graph's central nodes represent a tweet, while the other nodes represent retweets of the tweet, and the edges represent 'friend/follower' relationships between the nodes. These are called distribution graphs and we will describe them in the Section 4.2.

For Dataset 3, a large set of Covid-19 related Tweets are collected. A subsample of the collected tweets are extracted, specifically tweets with large graphs. The tweets are manually labeled into three classes: '5G Conspiracy', 'Other Conspiracy' and 'Non Conspiracy'. The graphs of the tweets are obtained in the same way as for Dataset 1 and Dataset 2.

4.2 Distribution graphs

Our data is collected from the social networking service Twitter. Twitter allows you to write and receive short messages that are called tweets. A user receives tweets by 'following' other users, creating a friendship between users. The friendships are directed, meaning that when a user is 'following' another user they are their 'friends', but not the other way around unless they are following them back. Tweets from friends are received in the message display called the timeline. A user has the option to comment and/or retweet other users tweets. A retweet is a re-posting of a tweet.

In this thesis we will be working with what we refer to as a distribution graph. A distribution graph is a subgraph of the Twitter follower network, that shows how a tweet is propagated on the platform by retweets. The distribution graphs are centered around a specific tweet that is associated with a user, and we will call this the central node. All other nodes represent retweets of other users, where the edges represent friend/follower relationships between the users. In an ideal distribution graph the edges would have represented the direction of the retweets, resulting in a propagation tree. However, based on how the data is collected from Twitter's API it is impossible to know the specific user someone has retweeted from. Twitter's API provides

only a list of users who have retweeted a tweet, and does not differentiate between a retweet of a tweet and a retweet of a retweet. The graphs can therefore have multiple edges directed outwards from a node and even have nodes that are disconnected from the graph, meaning that a user has retweeted without following the tweeter. A simple distribution graph is illustrated below in Figure 6.

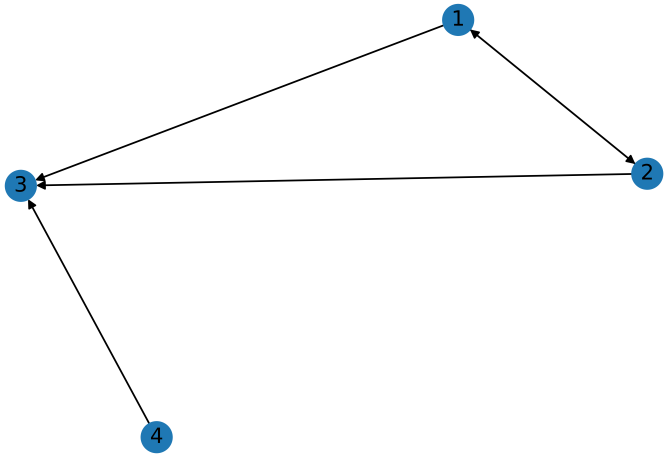

Figure 6: A simple distribution graph with four nodes labeled with arbitrary numbers and five directed edges.

In the simple example in Figure 6 it is easy to see that the central node is most likely the node with the node label '3'. In the distribution graphs we are assuming that the users are retweeting from their friends, but in theory any other node than node '3' can potentially be the central node since it is possible for a user to retweet a user without following that user. Notice also that node '1' and node '2' are friends of each other, making it is impossible to predict if they both retweeted directly from node '3' or if only one of them retweeted from node '3' and then the other retweeted via its friend.

In many cases we have missing data in the distribution graphs, e.g. missing nodes and links, making the graphs incomplete. This can make it impossible to identify the central nodes.

4.3 PyTorch Geometric and Data Format

Our datasets are in the same format as the *Benchmark Data Sets for Graph Kernels* [45] from TU Dortmund and we will refer to this as the TU Dortmund format. The *Benchmark Data Sets for Graph Kernels* consists of various datasets that are used for benchmarking in the field of graph kernels and graph neural networks. The TU Dortmund format is described as follows (where n = total number of nodes, m = total number of edges, and N = number of graphs):

- DS_A.txt (m lines): sparse (block diagonal) adjacency matrix for all graphs, each line corresponds to (row, col) resp. (node_id, node_id). All graphs are undirected. Hence, DS_A.txt contains two entries for each edge.
- DS_graph_indicator.txt (n lines): column vector of graph identifiers for all nodes of all graphs, the value in the i-th line is the graph_id of the node with node_id i
- DS_graph_labels.txt (N lines): class labels for all graphs in the data set, the value in the i-th line is the class label of the graph with graph_id i
- DS_node_labels.txt (n lines): column vector of node labels, the value in the i-th line corresponds to the node with node_id i

For the actual GNN modelling, we will be using the GNN framework PyTorch Geometric [6]. PyTorch Geometric is an extension to the popular deep learning library PyTorch [7]. It allows for deep learning on graphs. The implementation of GNNs with PyTorch Geometric will be described in Chapter 5, but we will here describe how PyTorch Geometric handles graph data.

In order to train a GNN with PyTorch Geometric, our data needs to be converted from the TU Dortmund format to the graph format used in PyTorch Geometric. PyTorch Geometric describes a graph by an instance of *torch_geometric.data.Data*, which includes the following attributes:

- data.x: Node feature matrix with shape [num_nodes, num_node_features]
- data.edge_index: Graph connectivity in COO format with shape [2, num_edges] and type torch.long
- data.edge_attr: Edge feature matrix with shape [num_edges, num_edge_features]
- data.y: Target to train against (may have arbitrary shape), e.g., node-level targets of shape [num_nodes, *] or graph-level targets of shape [1, *]
- data.pos: Node position matrix with shape [num_nodes, num_dimensions]

PyTorch Geometric has a class called *TUDataset*, that allows one to collect the benchmark datasets from TU Dortmund University's website and converts the data into the PyTorch Geometric format described above. We have slightly modified the TUDataset class in order to convert and use our own data. The data is uploaded to a Github repository where the data is collected and converted when needed.

4.4 Dataset 1: Small Politician Dataset

Dataset 1 consists of distribution graphs from politicians from four different German political parties. The parties are; 'Alternative für Deutschland (AfD)', 'Christlich Demokratische Union (CDU)', 'Die Linke' and 'Die Grünen'. The distribution graphs do not contain any node or edge features, other than a node label that is unique for each user. The node labels are discarded, because they will reveal the class of the distribution graphs when doing classification, which defeats the purpose of the classification.

We want to have graphs from politicians that have different types and sizes of follower networks, resulting in a diverse dataset. Based on this, four politicians in each party were selected as the source for the distribution graphs. After the data has been collected from Twitter, the politicians in the same parties are grouped together and given the same label. The individual politicians are listed in Table 1 below.

Die Linke	**Die Grünen**	**AfD**	**CDU**
Katja Kipping (3435)	Ricarda Lang (1457)	Stephan Brandner (3765)	A. Kramp-K. (1690)
Bernd Riexinger (2900)	Michael Kellner (1331)	Alice Weidel (1568)	Paul Ziemiak (1461)
Gregor Gysi (1313)	Maike Schaefer (1073)	Jörg Meuthen (1360)	Tobias Hans (876)
Nicole Gohlke (957)	Annalena Baerbock (698)	Tino Chrupalla (256)	
Total: 8605	**Total: 4559**	**Total: 6949**	**Total: 4027**

Table 1: Twitter sources from Dataset 1. The number after each politician is the respective number of graphs.

As seen in Table 1, the number of graphs per politician varies a lot. This is due to the fact that some politicians are more active, in the sense of tweeting, than others. Initially, the collected dataset had a much greater number of graphs than the numbers listed in Table 1, but it turned out that the majority of the graphs only contained one node. This is because a lot of the tweets are old which makes it difficult to collect the retweets, and in a lot of the cases the tweets simply do not have any retweets. The graphs with one node were discarded, since they do not contain any retweets. Table 1 also shows that 'CDU' is the only party with three politicians instead of four. It turns out that one of the politicians in 'CDU' has never tweeted and was discovered after the data had been collected.

Besides the variation in the number of graphs each politician contributes, the size of the individual graphs varies a lot as well. Size refers to the total number of nodes in each graph. The number of nodes in a graph, is essentially the number of retweets subtracted by one (the central node). The number of retweets does not just depend on the number of followers a user has, but also how active the followers are, the content of the tweet, the time the tweet was posted etc. To further investigate the varying size of the graphs, we have plotted size distributions for each party. The plots can be seen in Figure 7. On the horizontal axis the number of nodes is represented while the vertical axis represents the number of graphs.

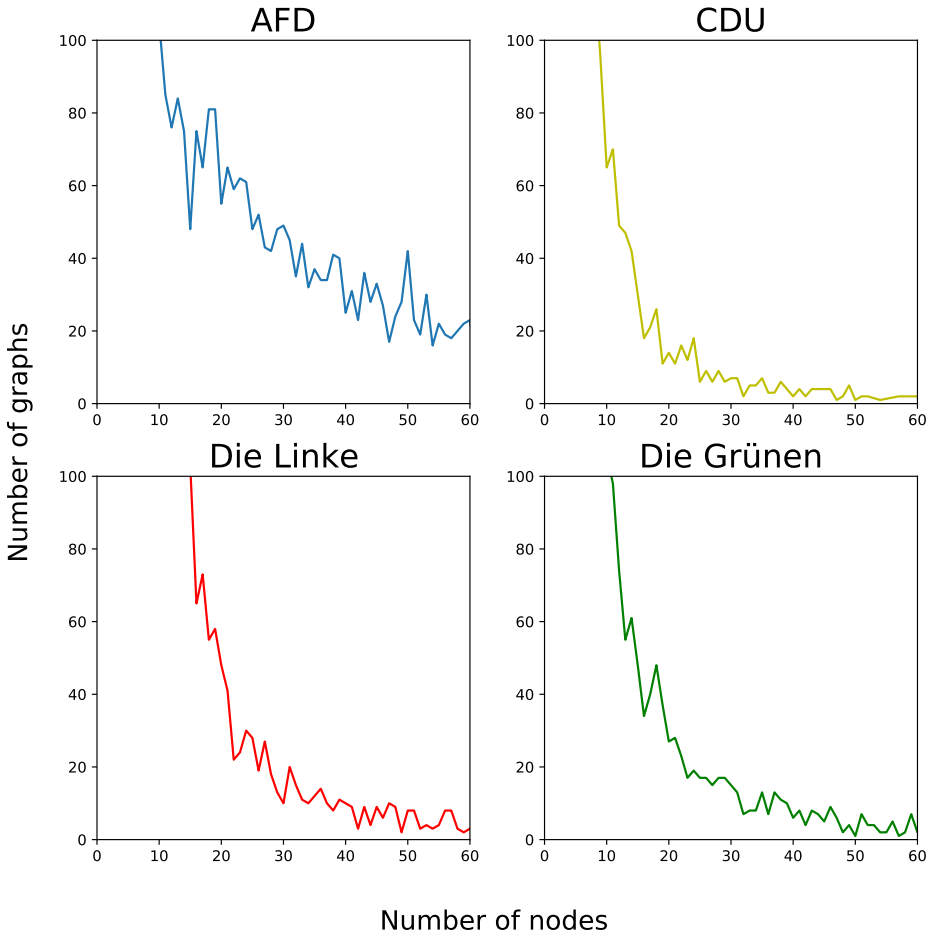

Figure 7: Distribution of graph sizes for each party for Dataset 1. The horizontal axis represents the number of nodes, while the vertical axis represents the number of graphs.

Figure 7 shows that the number of graphs are decreasing with respect to the size of the graphs. This means that smaller graphs (fewer retweets) are more common than larger graphs (more retweets). This can be a problem when doing classification since smaller graphs contain less structural information than larger graphs. Figure 7 also reveals that the distributions are different between the parties. For instance, 'AfD' is the only party that has a decent amount of larger graphs (above 50 nodes). When performing early experiments, we initially got a very high accuracy for certain classification tasks. It is reasonable to assume that the distribution of graph size had a direct influence on the model's ability to assign correct labels to the data. It is important that the GNN is not taking advantage of the difference in size to do the classification. By training a model to classify graphs based on the size, we are not distinguishing the graphs based on the structural information in the graphs. It would be an easy task to classify graphs from two different twitter sources if one had a massive follower network while the other one had a minor one. A way to avoid this, is by filtering out graphs of a certain size with a lower and upper bound. This enables us to be sure that all the data is of the same size. Filtering out small graphs is a sensible choice regardless, since small graphs do not contain any structural information whatsoever. On the other hand, the bigger the graphs are, the more structural information they contain. By simply discarding graphs with an upper bound we lose a lot of information. However, most of the classes in the dataset have a small amount of large graphs, thus by discarding large graphs we will not discard that many graphs compared to the overall

size of the dataset. A way to avoid discarding the large graphs, is by excluding them from the training process but to include them when evaluating the model. This would test if the GNN is able to generalize over the size of the graphs by evaluating the ability to classify graphs of sizes it has not trained on. However, as mentioned above, the limited amount of large graphs makes this difficult to evaluate.

We will set the lower and upper bound 20 and 40 nodes respectively. Meaning that every graph that is outside this region is discarded. In Table 2 the same politicians are listed as in Table 1, but after filtering the data with the lower and upper bound.

Die Linke	**Die Grünen**	**AfD**	**CDU**
Katja Kipping (99)	Ricarda Lang (104)	Stephan Brandner (117)	A. Kramp-K. (86)
Bernd Riexinger (97)	Michael Kellner (50)	Alice Weidel (336)	Paul Ziemiak (52)
Gregor Gysi (142)	Maike Schaefer (28)	Jörg Meuthen (390)	Tobias Hans (2)
Nicole Gohlke (5)	Annalena Baerbock (93)	Tino Chrupalla (28)	
Total: 342	**Total: 275**	**Total: 871**	**Total: 142**

Table 2: Twitter sources from Dataset 1 after filtering the data with a lower and upper bound of 20 and 40 nodes respectively. The politicians are order in the same manner as in Table 1, but the ordering of size has changed.

The overall number of graphs decreases drastically after filtering. As mentioned above, this is because the dataset initially contains a lot of small graphs. Even after the filtering, the size distribution will not be equal for all parties, but limiting the range of different sizes makes it more difficult for the GNN to simply rely on graph size to do classification.

Table 2 reveals that some politicians like 'Nicole Gohlke' and 'Thobias Hans' are underrepresented compared to the other politicians. However, if there is a strong link between the structure of the graphs and the parties, the GNN might be able to assign correct labels to underrepresented politicians based on the graph characteristics associated with a party.

To investigate the properties of Dataset 1 further, we have plotted the distribution of graph diameter in Figure 8.

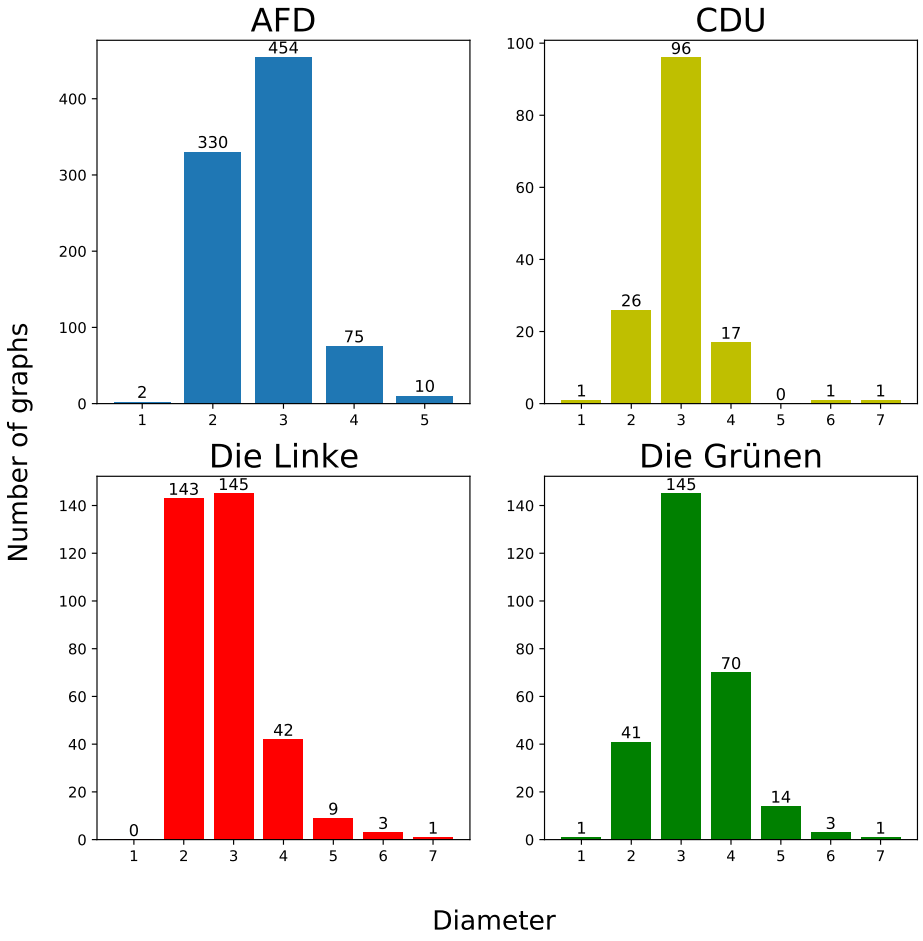

Figure 8: Diameter distribution of each party for Dataset 1. The horizontal axis represent the number of diameter, while the vertical axis represent the number of graphs.

The diameter tells us something about the complexity of the graphs and how far it spreads. Although, the number of layers of the graph would preciously tell us how far a tweet has spread, the diameter gives us an indication of it. Besides, the number of layers can only be computed for a tree structured graph, which the distribution graphs are not.

From Figure 8 we can see that the majority of the graphs, for all classes, has a diameter of 3. However, there still are some differences in the distribution. For instance, 'CDU' and 'Die Grünen' has a quite similar ratio between diameter 2 and 3, while 'Die Linke' has almost the exact same number of graphs of diameter 2 and 3.

4.5 Dataset 2: Large Politician Dataset

In Dataset 2 the graphs are still from the same German political parties as in Dataset 1, but the number of individual politicians is greatly increased to 180 politicians. Because of its large size, we will not name every politician that is included, but Table 3 summarizes some of the properties of Dataset 2.

	Die Linke	**Die Grünen**	**AfD**	**CDU**
Number of sources	12	29	73	66
Average number of graphs	298	386	1285	887
Total number of graphs	3579	11184	93807	58569

Table 3: The first row of the table indicates how many sources/politicians each party contains. The second row indicates the average number of graphs per source a given party has. The last row is the total number of graphs per party.

Table 3 shows that Dataset 2 has a lot more sources and graphs compared to Dataset 1. 'AfD' is again the majority class and is also the class with the most individual sources. To investigate further, we will again plot the distribution of graph sizes for the four parties.

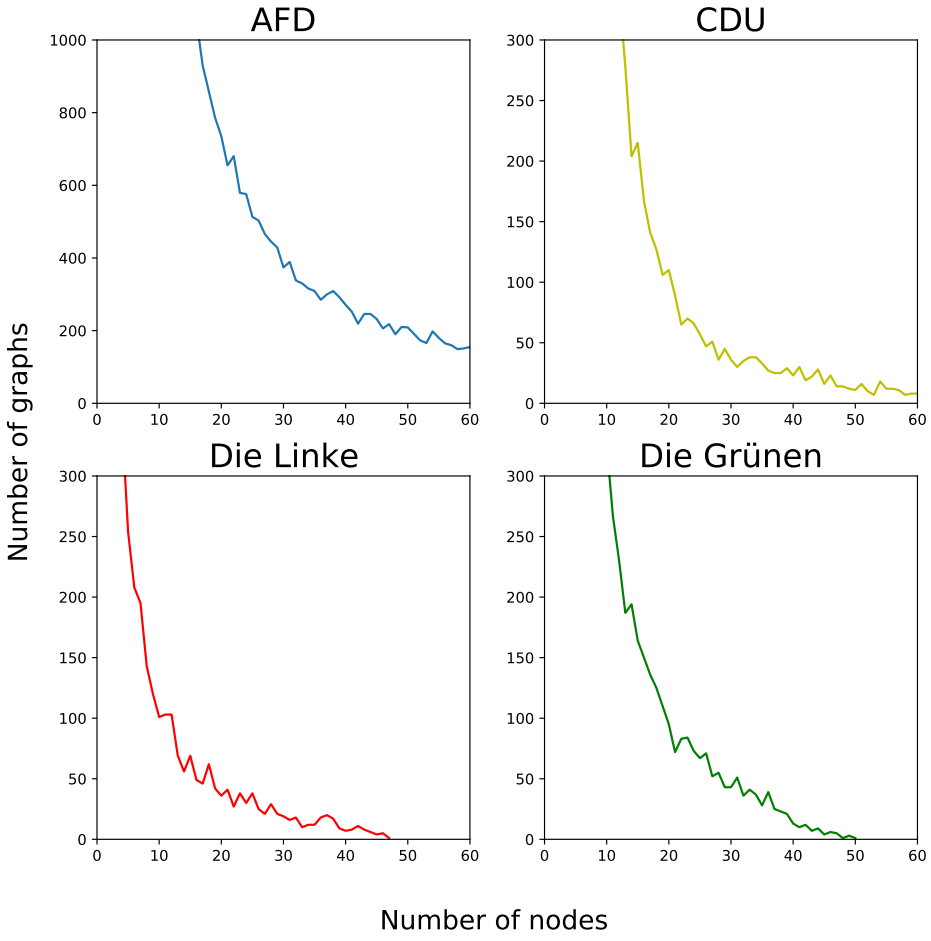

Figure 9: Distribution of graph sizes for each party for Dataset 2. The horizontal axis represents the number of nodes, while the vertical axis represents the number of graphs.

The distribution plots in Figure 9 look quite similar to the ones in Figure 7 in the sense that the number of graphs is rapidly decreasing with respect to the graph size. As before, we perform filtering of the graphs to smooth out the difference in sizes and to discard the smaller graphs. For Dataset 2, a different lower and upper bound is used than in Dataset 1. The lower and upper bound is selected to 15 and 35 respectively, in order to have a more even class distribution of the parties. Table 4 lists the same properties as in Table 3 after the data has been limited to only include graphs with 15-35 number of nodes.

	Die Linke	**Die Grünen**	**AfD**	**CDU**
Number of sources	11	29	69	50
Average number of graphs	99	97	290	78
Total number of graphs	1093	2820	20004	3901

Table 4: Properties of Dataset 2 after filtering with a lower and upper bound. The first row of the table indicates how many sources/politicians each party contains. The second row indicates the average number of graphs per source a given party has. The last row is the total number of graphs per party.

Since we have more sources than Dataset 1, we also have more variation in the number of graphs per source. Just as for Dataset 1, this can make it challenging for the classifier, because data is limited for certain politicians. Because of the large number of sources, instead of a table we visualize the number of graphs per source in a histogram for each party. The histograms can be seen in Figure 10.

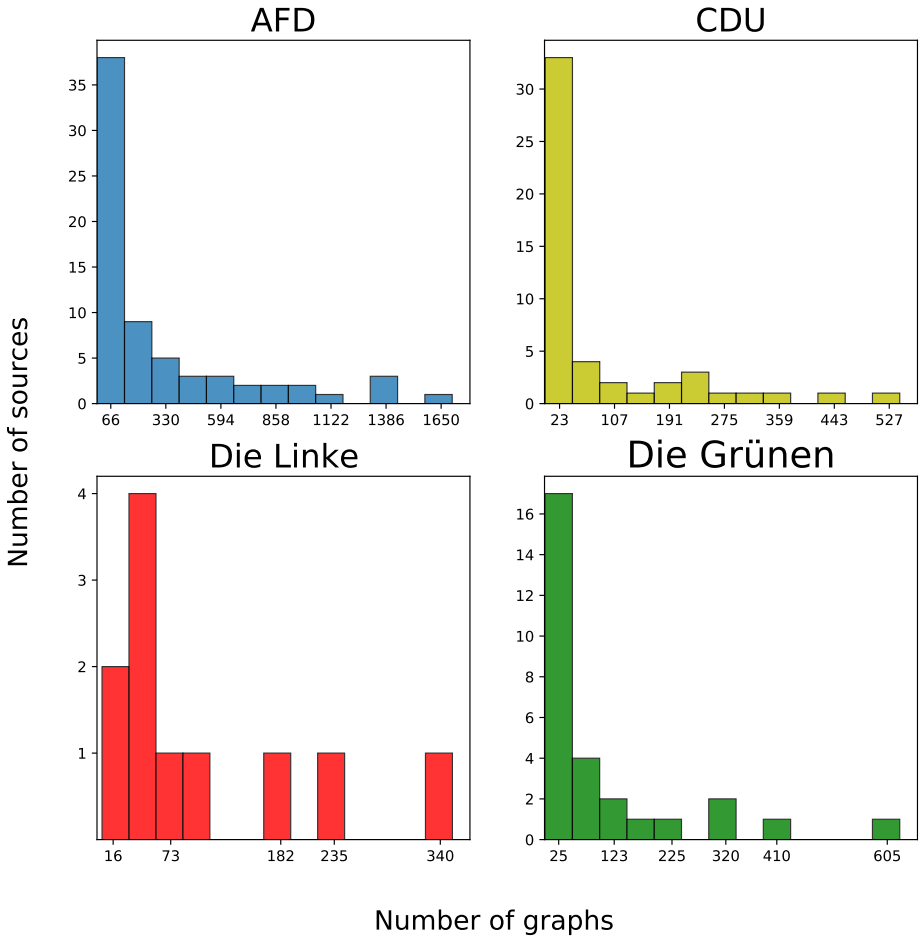

Figure 10: A histogram plot of the sizes of the source of Dataset 2. The horizontal axis represents the number of graphs, where each bin is divided into regions of graph sizes, and the vertical axis represents the number of sources.

Figure 10 shows that the majority of sources is concentrated in the first couple of bins, which means that most sources have a low number of graphs. There are only a few sources that have a large number of graphs and these make up the largest contribution of graphs for each class. The reason lies in the fact that most of the politician are passive users, meaning that they do not post a lot of tweets. This does partially defeat the purpose of Dataset 2, since we are increasing the number of sources from Dataset 1 with the intention of having more variety of individual politicians. On the other hand, the large number of passive sources sums up to a moderate size of graphs. Even though they might be underrepresented, the model might be able generalize over the sources in the party.

As with Dataset 1, we plot the distribution of graph diameter to get a better insight in the data. The diameter plot can be seen in the Figure 11 below.

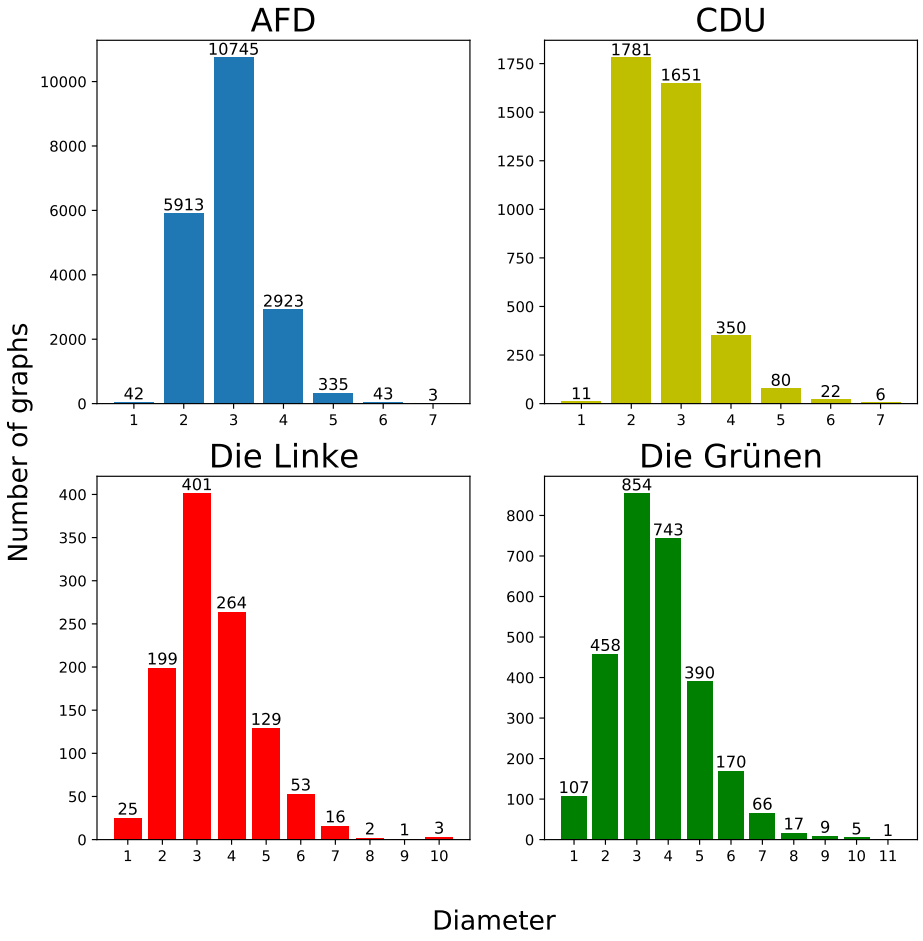

Figure 11: Diameter distribution of each party for Dataset 2. The horizontal axis represent the number of diameter, while the vertical axis represent the number of graphs. Please notice that the y-axis for 'AfD' is different than the other classes.

Figure 11 shows a different distribution of diameter for Dataset 2 compared to Dataset 1. 'CDU' is the only party that has 2 as the majority diameter. Furthermore, 'Die Linke' and 'Die Grünen' have relatively large diameters compared to the other two classes. 'Die Grünen' has also 107 graphs that has diameter equal to 1. This means that the graphs are fully connected, i.e. there is an edge between each node in the graph.

4.6 Dataset 3: Fake news

The final dataset, Dataset 3, consists of tweets that are related to COVID-19. Dataset 3 was collected by text search on COVID-19 related keywords. The data was manually labelled by a large group of scientists, postdocs and graduate students. Dataset 3 is categorized into three classes; '5G Conspiracy', 'Other Conspiracy' and 'Non Conspiracy'.

The class '5G Conspiracy' contains tweets that are drawing causal links between COVID-19 and the 5G cellular networks. This includes theories such that the pandemic is caused by 5G's weakening effect on the immune system, or that the victims of COVID-19 are harmed by the radiation from the 5G network towers, denying the existence of the virus.

'Other Conspiracy' contains tweets about conspiracies related to COVID-19 that are not 5G related. The tweets are primarily from large organization and state actors that are spreading misinformation often with the intent of distracting the public from other events. Some of the topics in this class are e.g. governments intentional release of COVID-19, harmful vaccinations and the virus being a hoax.

The last class, 'Non Conspiracy', contains tweets that do not fit in the other two classes. This includes tweets that show scepticism towards 5G technology, but do not link 5G with COVID-19, and tweets that are commenting or being sarcastic about the conspiracies that are mentioned above.

4.6.1 Node features

In Dataset 3 we have additional information in the form of node features. Each node in the graphs has a timestamp as feature value. The timestamps are the response time from when a retweet is posted after the original tweet. The timestamp for the central node is therefore '0' since it is the original tweet.

The node features help build better distribution graphs. It reveals the central node, which is not always a trivial task solely based on the structure. It can also in certain scenarios identify who an user retweeted, if the node has several outgoing edges. To clarify this, let us bring back the distribution graph in Figure 6, but this time we will include timestamps. The timestamps are written next to the arbitrary node labels and are written in seconds.

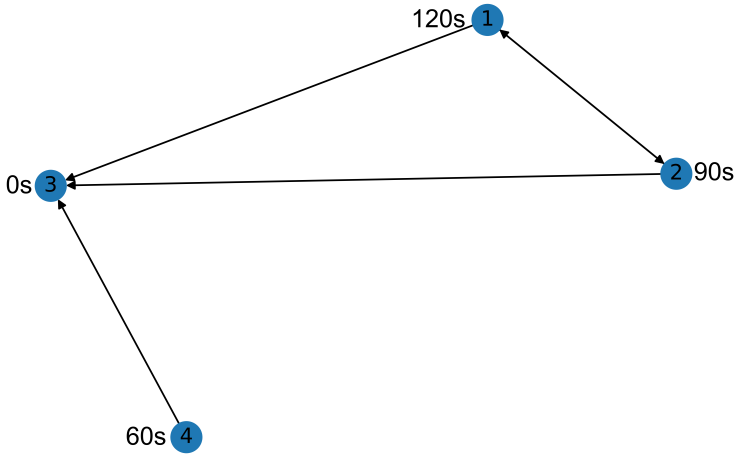

Figure 12: The distribution graph from Figure 6 with timestamps. The timestamps are written in seconds next to the node labels. The arbitrary node labels are just for illustration purposes only, the real node labels or node features are the timestamps.

With the timestamps as seen in Figure 12, we can say with certainty that node '3' is the central node of the graph. It is still not possible to say whether node '1' retweeted from node '2' or node '3', but based on the timestamps we can say for certain that node '2' retweeted directly from node '3', since node '1's timestamp is of higher value than node '2's.

The GNNs that we are using throughout the work of this thesis, handle node features as labels. The timestamps are continuous variables, and we therefore have to convert them to labels. The timestamps are converted into four categories that were chosen based on the distribution plot of the timestamps. The distribution plot can be seen below in Figure 13.

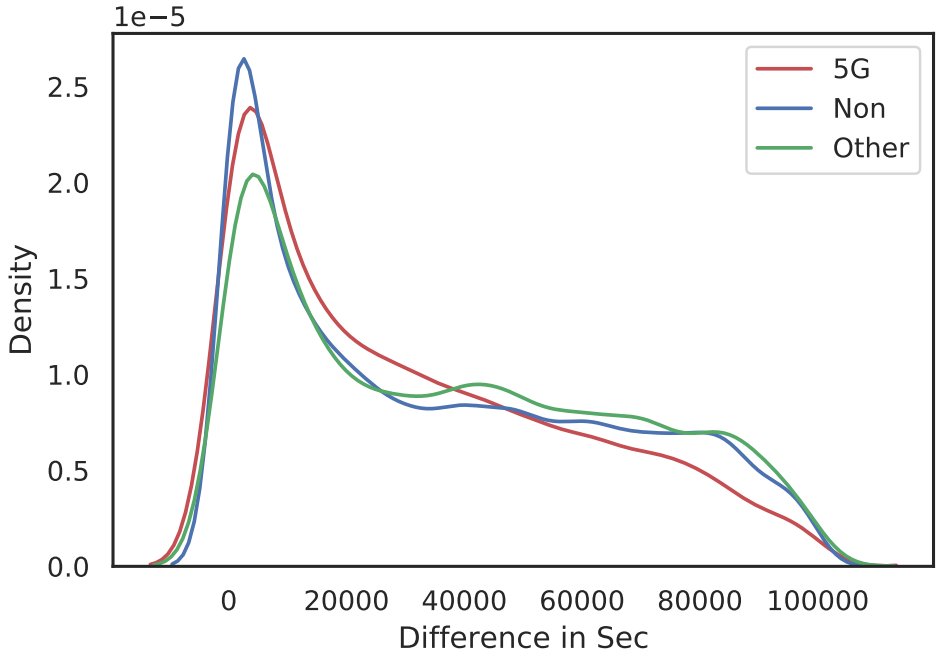

Figure 13: The distribution of timestamps of Dataset 3. The horizontal axis shows the the timestamps in seconds and the vertical axis is the density of the estimated distribution.

As seen in Figure 13, the distribution of timestamps between the classes are seemingly alike.

The four categories the timestamps are converted to are listed below (where t is timestamp).

- **0** for $t = 0s$
- **1** for $0s < t < 20.000s$
- **2** for $20.000s < t < 60.000s$
- **3** for $60.000s < t$

As with Dataset 1 and Dataset 2, we will plot the size distribution plots of each class in Dataset 3. The distribution plots are illustrated in Figure 14 below.

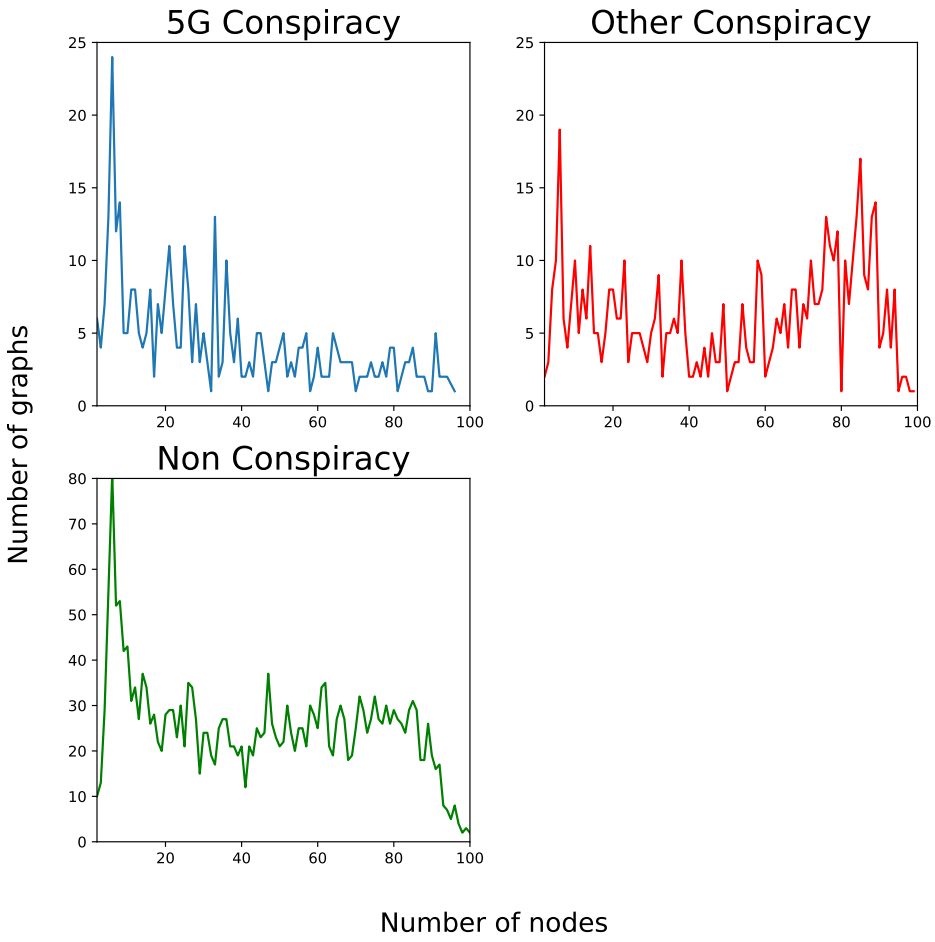

Figure 14: Distribution of graph sizes for each party for Dataset 3. The horizontal axis represents the number of nodes, while the vertical axis represents the number of graphs. Please notice that the y-axis for 'Non Conspiracy' is different from the two other classes.

We can see in Figure 14 that the graph sizes in Dataset 3 are more evenly distributed in each class compared to Dataset 1 and Dataset 2. Dataset 3 also has a fair amount of larger graphs for all classes. This allows us to include all larger graphs, without having to set an upper bound. We still need to set a lower bound to filter out smaller graphs due to the absence of structural information as mentioned before. Like in Dataset 1, we will set the lower bound limit to 20 number of nodes.

As before, we will plot the diameter distribution plots of each class after we have filtered out the small graphs. The plots can be seen in Figure 15 below.

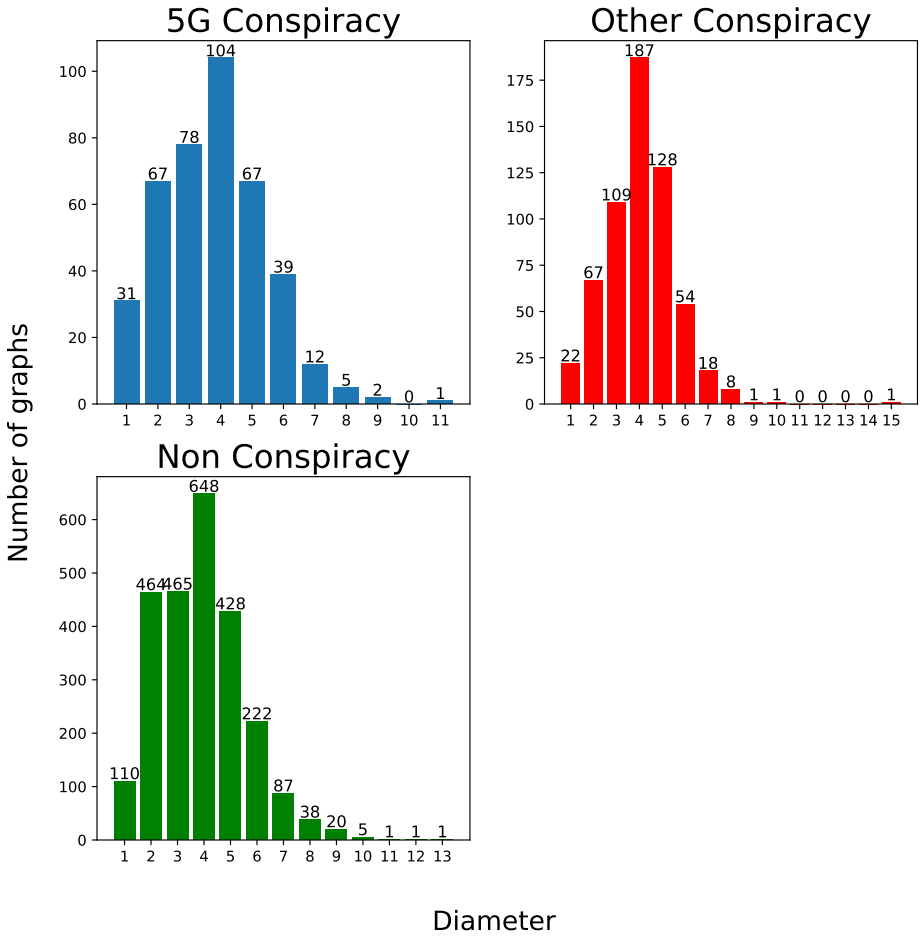

Figure 15: Diameter distribution of each class for Dataset 3. The horizontal axis represent the number of diameter, while the vertical axis represent the number of graphs.

From Figure 15 we can see that Dataset 3 has graphs with bigger diameters compared to the other two datasets (as can be seen in Figure 8 and 11). The majority of the graphs has diameter equal to 4. Each class has almost the identical distribution of diameter, which makes it difficult to use as a feature for classification.

5 Experimental Setup

This chapter gives a detailed outline of the setup of the experiments prior to any modelling. The first section, Section 5.1, gives an overview over the different GNN models that are chosen for the experiments. Section 5.2 describes the implementation of the models. Section 5.3 gives a detailed description of how the experiments are defined; specifying the classification tasks and how the training process is defined. The last section, Section 5.4, describes how the performance of the models is evaluated.

5.1 Models

As mentioned in the previous chapters there exists numerous types of GNNs. We will specifically look at Spatial-based ConvGNNs for graph classification. Based on its promising results using only structural information (no node or edge features) our main model will be the Graph Isomorphism Network (GIN) [21]. We will also do a broader comparison of different state-of-the-art GNNs based on the paper written by Errica et al. [46]. As a baseline model we will use a random forest (RF) classifier.

5.1.1 Baseline: Random Forest Classifier

A baseline model is used to compare the GNN models to a more naive and simpler method. We will use a random forest classifier [47] as a baseline model. Simply put, random forest uses multiple decision trees that are merged together to do prediction. Before training the classifier, features have to be extracted manually from the data. The features are what define the power of the classifier. If the features are describing the graphs well, the classifier will have good results. The features that are used are the following:

- Number of nodes
- Number of edges
- Graph diameter
- Top 6 of the highest node degrees (used as 6 separate features)
- The mean clustering coefficient

Even though we have filtered out graphs to have a even distribution of graph sizes, the distribution is still not uniform. Number of nodes is therefore included as a feature to allow the random forest model to take advantage of this.

5.1.2 GIN-Baseline

GIN is a simple GNN architecture, but it is proven to be one of the most expressive architectures among the GNNs [21]. Since we are only performing graph classification, we are using the pooling variant of GIN to get a graph representation of the computed node embeddings. For a detailed description of GIN, the reader is referred to Section 2.3.4. We will refer to our implementation of the GIN model as the GIN-Baseline model.

5.1.3 GNN Comparison Framework

In order to compare the experimental results of the GIN-Baseline with other state-of-the-art GNN models, we use a GNN comparison framework proposed by Errica et al. [46]. They compare different state-of-the-art GNN architectures for graph classification on common benchmark datasets. They claim that most published works and results in the graph representation learning field are in many cases ambiguous or not reproducible, especially concerning hyperparameter selection and correct usage of data split in terms of model selection versus model assessment. The framework includes the following GNN architectures and are described in more detail in Section 2.3.4 :

- Deep Graph Convolutional Neural Network (DGCNN) [26]
- DiffPool [25]
- Edge-Conditioned Convolution (ECC) [24]
- Graph Isomorphism Network (GIN) [21]
- GraphSAGE [18]

As seen in the list of GNNs above, GIN is also included in the framework. This makes it possible for us to do a fair comparison between GIN and the other GNNs in the list, because the training and evaluation environment is equal for all the GNNs in the framework.
Additionally to the paper [46], they provide an open access library on Github that makes it easy to reproduce the experiments in the paper. The library is not limited to work on the benchmark datasets. With small modifications to the library provided, we are able to implement the comparison framework with our own datasets.

5.2 Implementation

The models are implemented using Pytorch Geometric [6]. As mentioned in Chapter 4, Pytorch Geometric is an extension library to Pytorch [7] that allows for deep learning on graphs. Pytorch Geometric remarkable speed and well implemented GNN models from various papers, make it a state-of-the-art GNN framework and well suited for this thesis. Pytorch Geometric has all the GNN architectures that are mentioned in Section 5.1 already built in, making the implementation of the experiments relatively straightforward.

Google Colab [48] is used as an environment to build and execute the GIN-Baseline. Google Colab provides free use of GPUs and supports CUDA operations, and can therefore be used with Pytorch to reduce the execution time.

The framework provided by Errica et al. [46] is also using Pytorch Geometric. Before running our experiments with the comparison framework, the framework has to be modified first in order for us to use our own datasets.

Running the experiments with the comparison framework is extremely time consuming because of its excessive grid search of different configurations for each model (See Section 5.3.5). Because of this, we are using Simula's eX^3 cluster [49] to run the experiments. eX^3 is a cluster that contains large varieties of High Performance Computing (HPC) hardware. This allows us to run all the GNN models in the framework simultaneously on different nodes in the cluster.

5.3 Experiments

5.3.1 Classification tasks

The overall objective of the experiments is to examine the possibility of identifying and distinguishing different spreading graphs on Twitter. The classification tasks are solved as a binary classification problem as well as a multiclass classification problem. Throughout the experiments, for each dataset the data typically remains the same but the labeling of the graphs changes. A more detailed description of the classification tasks for each dataset is given below.

German Politicians: Dataset 1 and Dataset 2

For Dataset 1 and Dataset 2, we want to explore if there is a correlation between political parties and the characteristics of the Twitter graphs. Besides classifying individual parties as a multiclass problem, we examine the possibility to classify parties based on where they are seated in the German federal parliament (The Bundestag) [50]. The allocation of seats in the parliament can be seen in the Figure 16 below.

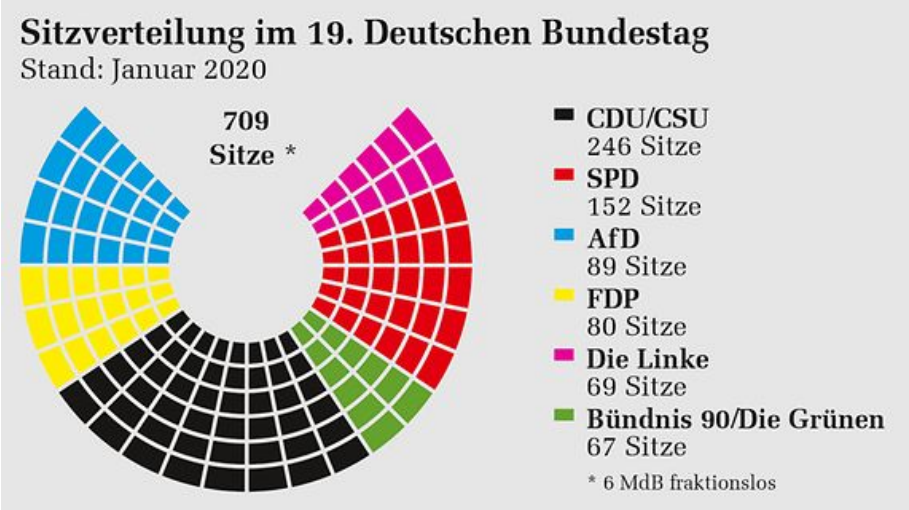

Figure 16: Illustration from DBT [50]. The figure illustrates the allocation of seats in the German parliament. The colours represent the different parties; where 'AfD' is blue, 'CDU' is black, 'Die Grünen' is green and 'Die Linke' is purple.

For the four parties included in Dataset 1 and Dataset 2, Figure 16 shows that from the right (left in the figure) we have 'AfD', 'CDU', 'Die Grünen' and 'Die Linke'. Based on this, we will group the parties into two classes to form a binary classification problem.

'AfD' is much more active on Twitter than the rest of the parties, which means that we have a much greater number of graphs from 'AfD'. Since 'AfD' is the rightmost party and the majority class, it makes sense to group 'AfD' alone for the first classification tasks. In the second classification task we want to see the GNN works under different aggregation of multiple parties by making groups of two parties together. The last classification task is the most natural scenario where we simply classify the given parties resulting in a multiclass problem. All three classification tasks for Dataset 1 are listed below:

Dataset 1 Experiments:
'AfD' vs 'CDU' and 'Die Grünen' and 'Die Linke' (Far right vs other parties)
'AfD' and 'CDU' vs 'Die Grünen' and 'Die Linke' (Right vs Left)
'AfD' vs 'CDU' vs 'Die Grünen' vs 'Die Linke'

As described in Chapter 4, Dataset 2 includes graphs from the same parties as in Dataset 1, but from a much greater number of individual politicians. Performing the same experiments on Dataset 2 as in Dataset 1, gives an indication of how robust our results are in relation to the number of sources.

For Dataset 2, the data from 'Die Linke' was much smaller in both overall number graphs and in number of nodes per graph compared to the rest of the parties. It was therefore excluded from the experiments on Dataset 2.

To be consistent with the experiments in Dataset 1, the classification tasks for Dataset 2 are the same as for Dataset 1. However, since 'Die Linke' is not included, the experiments are not quite comparable. For instance the second classification class makes less sense since we now have two parties vs one party instead of two parties vs two parties. However, this can be seen as a variation of the first classification task where we have one party vs the rest. The classification tasks for Dataset 2 are listed below:

Dataset 2 Experiments:
'AfD' vs 'CDU' and 'Die Grünen' (Far right vs Not far right)
'AfD' and 'CDU' vs 'Die Grünen' (Right vs Left)
'AfD' vs 'CDU' vs 'Die Grünen'

Fake news: Dataset 3

For the last dataset, we want to explore the possibility of identifying fake news based on the spreading graphs. As described in Chapter 4, we have three different categories of Tweets in Dataset 3: '5G Conspiracy', 'Other Conspiracy', and 'Non Conspiracy'.

As in the case of Dataset 1 and Dataset 2, we are mainly focusing on the structure of the graphs alone, but we also do an additional experiment with added node features.
The three different classification tasks for Dataset 3 are listed below:

Dataset 3 Experiments:
5G Conspiracy vs Other Conspiracy vs Non Conspiracy
5G conspiracy vs Non Conspiracy
Other Conspiracy vs Non Conspiracy

5.3.2 Node features

In the original GIN paper [21], when node features are not given, they use one-hot encodings of node degrees as feature values when modelling on social graphs. They do not give reasoning for this particular modelling choice. GNNs that are based on message passing should be able to infer this structure information without specifying it as a feature. However, Errica et al. [46] proves that using node degrees as features are in many cases beneficial in terms of performance and reduces the number of GNN layers needed to achieve good results.

As described in Chapter 4, Dataset 1 and Dataset 2 does not contain any node features. We will therefore use one-hot encodings of node degrees in every experiment for the GIN-Baseline. The comparison framework includes the use of node degrees in the hyperparameter optimization (more on this in Section 5.3.6). For Dataset 3 we will compare the use of timestamps as node features vs. node degree as node features.

5.3.3 Mini-batching

As described in Section 2.1, the use of mini-batching in deep learning is an important technique to enable use of large amounts of data.

The mini-batching approaches that are typically used with Euclidean data do not hold for graphs because of graphs' highly irregular structure. It is neither feasible or memory efficient when using graphs. PyTorch Geometric solves this by stacking the adjacency matrices in a diagonal fashion, i.e. creating a big graph that holds multiple isolated subgraphs, and concatenates the node and target features in the node dimension. Mini-batching in PyTorch Geometric is illustraded in Equation 24 below.

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_1 & & \\ & \ddots & \\ & & \mathbf{A}_n \end{bmatrix}, \qquad \mathbf{X} = \begin{bmatrix} \mathbf{X}_1 \\ \vdots \\ \mathbf{X}_n \end{bmatrix}, \qquad \mathbf{Y} = \begin{bmatrix} \mathbf{Y}_1 \\ \vdots \\ \mathbf{Y}_n \end{bmatrix}. \tag{24}$$

Equation 24 illustrates how mini-batching in PyTorch Geometric works, where A is the adjacency matrices stacked together, and X and Y are the node and target features concatenated together respectively. n is the size of the mini-batch. This approach still works with GNN that are using a message passing scheme, such as GIN, since nodes in different graphs can not exchange messages. It is also memory efficient; no extra padding of node or edge features is needed and the adjacency matrices are saved in a sparse fashion, i.e they are only holding non-zero entries.

5.3.4 Resampling

As mentioned in Section 4, the classes in the datasets are not uniformly distributed resulting in somewhat unbalanced sets. This obviously depends on how we are grouping/labeling the graphs, but regardless, the classes are never evenly distributed.

A way to handle class imbalance is by random resampling the imbalanced dataset resulting in a new transformed dataset. This is either done by undersampling the majority class or by oversampling the minority class, or a combination of the two. Undersampling is taking a subsample of the total data of a class and discarding the rest. This is done less frequently than oversampling, since it is only a sensible thing to do if one has sufficient data points, or/and if one wishes to reduce the computational cost of a model, due to the reduced amount of data. Oversampling is done by including duplicates of the minority classes in the training data. This can be done for all the instances in the minority classes or at random with or without replacement, where replacement means that they can be selected at random after they have been selected once. Oversampling, and especially with replacement, can result in overfitting on the training data since the model is trained multiple times on the same instances of data points. Both undersampling and oversampling are naive methods since they do not assume anything about the data. They are therefor simple to use and can be desirable for large complex data sets.

For the experiments on the political datasets we do not favour any classes, we are only aiming for the highest possible classification accuracy. In other words false positives and false negatives are not superior to each other. In the case of the fake news dataset, we might favour false negative over false positives, meaning that we rather want to classify a non-fake tweet as fake news (false negative) than a fake news tweet as non-fake news (false positive). This means that for the experiments on the political datasets it might not be as necessary to resample

the dataset, as opposed to the fake news dataset. Furthermore the political datasets are not drastically imbalanced, but there are differences in the distribution in the different experiments which makes it important to use a metric that gives a fair evaluation of the models no matter the distribution of the classes (more on this in Section 5.4).

Nevertheless, with Dataset 1 we will perform the experiments without resampling, this is because the classes are not as imbalanced as in Dataset 2 and Dataset 3. For Dataset 2 and Dataset 3 we will perform experiments only with balanced resampled datasets. For the resampling we perform random undersampling of the majority classes. The resampling of Dataset 2 and Dataset 3 is visualized in Figure 17 and 18 below.

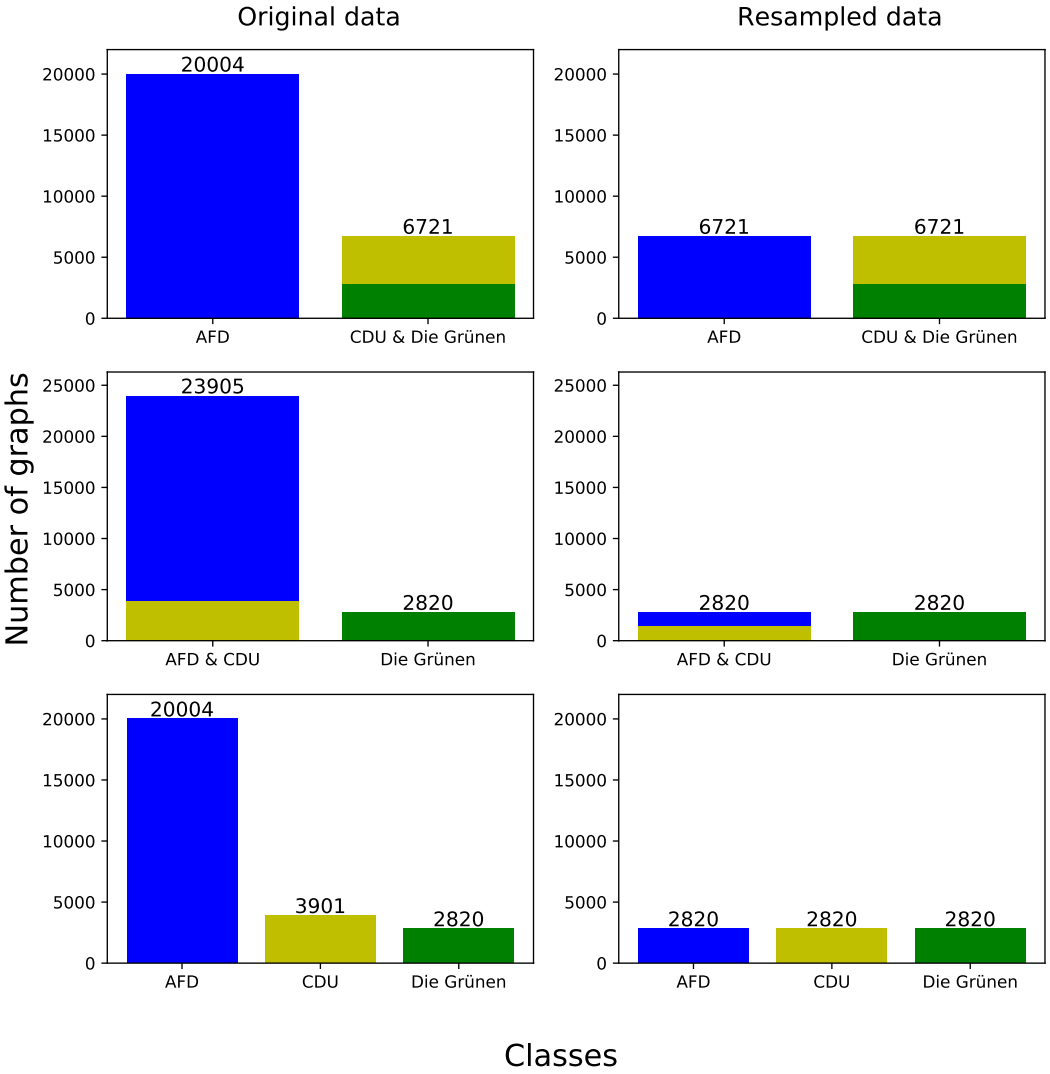

Figure 17: Resampling of Dataset 2 for the three classification tasks by undersampling of the majority class/classes. If the majority class consists of multiple classes the undersampling is done in such a way that the resampled data represent both classes equally.

Figure 17 illustrates how Dataset 2 is resampled for all classification tasks. The majority class/classes are undersampled to the size of the minority class, resulting in a balanced datasets.

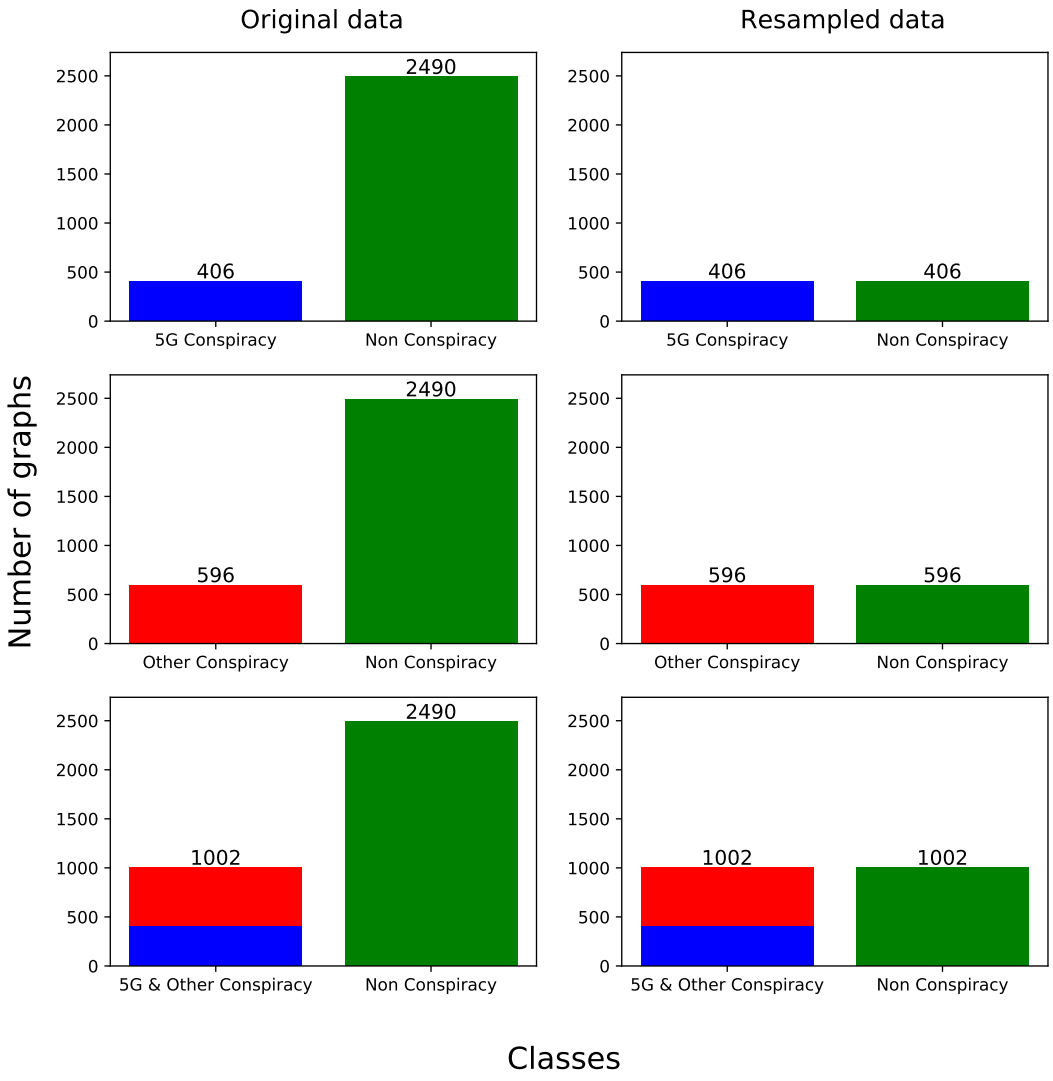

Figure 18: Resampling of Dataset 3 for the three classification tasks by undersampling of the majority class which is the class 'Non Conspiracy' for all tasks.

Figure 18 illustrates how Dataset 3 is resampled to balanced sets by undersampling the class 'Non Conspiracy' for all three classification tasks.

5.3.5 Cross validation

To validate the model, we use a k-fold cross validation. Cross validation is used to generalize a model to an independent data set. When data is limited, which it usually is, we are able to maximize the usage of the data we have. It gives an estimate on how accurately a predictive model will perform on unseen data. A k-fold cross-validation splits the datasets into k equal sized subsamples. For each k subsample, a subsample is selected as a test set while the remaining data are left as a training set. A model is then fitted to the training set and evaluated on the test set. The final evaluation of the model is the mean value of the k evaluation scores as well as the standard deviation of the folds.

GIN-Baseline
For the GIN-Baseline, we use a 10-fold stratified cross validation with a test, training and validation set. The pseudo-code for the cross validation algorithm can be seen below.

Algorithm 3 Cross validation

```
Input: Dataset D
Split D into 10 folds F_1, ..., F_k
for i ← 1, ..., 10 do
    train_i, test_i, val_i ← (∪_{k≠j∪i} F_k), (∪_{j≠i} F_j), F_i
    for e ← 1, ..., 100 do                                   ▷ e is epoch
        model_{i,e} ← train(train_i)
        EvalLoss_{i,e} ← eval_loss(model_{i,e},val_i)
        EvalScore_{i,e} ← eval_score(model_{i,e},test_i)
    end for
    SelectedEpoch ← argmin(EvalLoss_i)
    Loss ← EvalLoss_{i,SelectedEpoch}
    Score ← EvalScore_{i,SelectedEpoch}
end for
MeanLoss ← mean(Loss)
MeanLoss ← mean(Score)
```

The dataset is partitioned in 10 equal sized subsamples. The test and validation set are each a single subsample of the dataset, while the training set is the 8 remaining subsamples. For each fold, the network is trained on the training set, then a validation loss is derived from the validation set and finally an evaluation score is determined from the test set. After the model has trained for 100 epochs, we select the epoch that has the lowest validation loss. The corresponding evaluation score for this epoch is selected as the final score for the fold. The weights of the network are reinitialized after each fold to avoid leakage of information between the folds. The mean value of the evaluation scores of the 10 folds is then set as the final evaluation score.

With this method we prevent overfitting on either sets, since we are only using the validation set to stop the training when the loss is at its lowest, and the final evaluation of the model is derived from the unseen test set. By using a stratified cross validation, we are forcing the class distribution in each split to be equal, or approximately, to the true class distribution of the whole dataset. This helps avoiding a biased classification algorithm when using evaluation metrics such as accuracy that weights each instance equally. Even when using metrics that are unbiased, classes that are not well represented in a fold will be poorly learned by the model. This is prevented by the use of stratified folds where each class is well represented with respect to the true distribution.

GNN Comparison Framework

The comparison framework uses a different algorithm for their cross validation. They split the training and evaluation process into two separate parts; *model assessment* and *model selection*. The pseudo-code for the model assessment and the model selection are shown in Table 5.

Algorithm 1 Model Assessment (k-fold CV)

Input: Dataset $\mathcal{D}$, set of configurations Θ
Split $\mathcal{D}$ into k folds $F_1, \ldots, F_k$
for $i \leftarrow 1, \ldots, k$ **do**
    $\text{train}_k, \text{test}_k \leftarrow \left(\bigcup_{j \neq i} F_j\right), F_i$
    $\text{best}_k \leftarrow \text{Select}(\text{train}_k, \Theta)$
    for $r \leftarrow 1, \ldots, R$ **do**
        $\text{model}_r \leftarrow \text{Train}(\text{train}_k, \text{best}_k)$
        $\text{p}_r \leftarrow \text{Eval}(\text{model}_k, \text{test}_k)$
    end for
    $\text{perf}_k \leftarrow \sum_{r=1}^{R} \text{p}_r / R$
end for
return $\sum_{i=1}^{k} \text{perf}_i / k$

Algorithm 2 Model Selection

Input: train_k, Θ
Split train_k into *train* and *valid*
$\text{p}_\theta = \emptyset$
for each $\theta \in \Theta$ **do**
    model $\leftarrow \text{Train}(\text{train}_k, \theta)$
    $\text{p}_\theta \leftarrow \text{p}_\theta \cup \text{Eval}(\text{model}, \textit{valid})$
end for
$\text{best}_\theta \leftarrow \text{argmax}_\theta \, \text{p}_\theta$
return best_θ

Table 5: Pseudo-code for the model assesment and model selection in Errica et al. [46].

The model assessment uses a 10-fold cross validation for model assessments and an inner holdout technique with 90%/10% training/validation split. The model selection chooses the configuration that results in the highest validation score. The configurations are different hyperparameters such as number of layers, hidden units, learning rate etc. This will be described in detail in Section 5.3.5. After the model selection, the model is trained and evaluated three times on the train and test set. This is done three times to avoid unfavourable weight initialization of the test performances. The final test performance is the mean of the three performance scores. During this model assessment, 10% of the training data is randomly held out as a validation set to perform early stopping. Early stopping is implemented with a patience parameter n, where the training stops after n epoch if the validation performance does not improve.

It should be pointed out that all the data splits in the model assessment and model selection are precomputed. This means that prior to any modelling, each dataset is already partitioned in the necessary splits. This is done in order to do a fair comparison of the different GNN architectures, where each GNN is trained and evaluated on the exact same data.

5.3.6 Optimization

We have two different types of parameters in a neural network:

- **Hyperparameter**. Parameters that are set prior to training the neural network (e.g. number of hidden layers and hidden units)
- **Model parameter**. Parameters that are learned by the network during training. Also known as the weights of the neural network.

Hyperparameters can not be directly learned from a regular training process, they have to be set before training. However, hyperparameter optimization is in some sense a way to learn the optimal hyperparameters for a given model. Hyperparameter optimization (also known as hyperparameter tuning) aims at finding an optimal combination of hyperparameter for a given learning algorithm.

GIN-Baseline
For the GIN-Baseline with Dataset 1, we do not perform any kind of exhaustive hyperparameter search. We simply train a model with a few different combinations of hyperparameters for the first classification task. Since we are modelling on the same data for all the classification tasks in Dataset 1 (they only differ by labeling), we only perform hyperparameter optimization on the first classification task. We then pick the hyperparameters that have the best results on the first classification task, and use this set of hyperparameters for all the classification tasks. The following hyperparameters are used for hyperparameter optimization for the GIN-Baseline (based on the hyperparameters used in the GIN paper [21]).

- **# Hidden Layers:** [2,3,5]
- **# Hidden Units:** [32,64]
- **# Batch Size:** [32,128]

The hyperparameters above are the only parameters that we are varying during the hyperparameter optimization. Other hyperparameters that are being used, but are held constant, are listed below.

- **Dropout:** $k = 0.5$
- **Learning rate:** 0.02
- **Step-decay:** $50, 0.5$

Where step-decay means we are reducing the learning rate by 0.5 for every 50 epoch.

GNN Comparison Framework
The comparison framework does a much more thorough hyperparameter optimization. They call this the model selection, where a model with a given set of hyperparameters is selected based on the highest accuracy score. They criticize other authors in the field of Machine Learning, of documenting over-optimistic results from the model selection by not separating the model selection and model assessment. This is one of the main contributions of Errica et al. [46]; keeping the model selection separate from the model assessment, resulting in an unbiased evaluation of the model. Please note that this is also the case in the GIN-Baseline, where the test set and validation set is held separate and only contribute to the model selection and model assessment respectively.

The comparison framework uses the hyperparameter configurations that are used in the original papers of the respective models. For all models, this ranges from 32 to 72 different combinations of hyperparameters and is without a doubt the most computational expensive part of the framework.

5.4 Metrics

Metrics are used to evaluate the performance of a model. A common evaluation metric for classification problems is accuracy. Accuracy is simply the overall percentage of correct classifications. It works well for binary classification problems on balanced data sets. For datasets with uneven class distributions, the accuracy metric results in a bad representation of the overall performance of the model. A naive model that assigns the majority class to all instances can end up with a high accuracy score if the proportion of the majority class is large. For instance if 90% of the total samples are in the 'AfD' class, this would result in a 90% accuracy. Furthermore, the accuracy does not give any meaningful insights when evaluating a multiclass problem. It does not reveal what classes the model is assigning correctly/incorrectly, it simply returns the overall accuracy.

The **Rk statistic** [51] is a metric that measures the correlation coefficient between the observed and predicted classifications. It is a multiclass generalization of the Matthews correlation coefficient [52]. Matthews correlation coefficient is the discrete case of the Pearson's correlation coefficient, which is a measure of association for two binary variables. Rk ranges from -1 to 1, where 1 is a perfect prediction, 0 a random prediction and -1 a completely false prediction. The Rk statistic considers true and false positives and negatives, resulting in a balanced measure even when the class imbalance is large.

As discussed previously our datasets are not balanced, but with random resampling we are forcing class balance. The Rk statistic is mainly used with Dataset 1 when not using resampling. With Dataset 2 and Dataset 3, we only train on balanced datasets and can therefore use accuracy as the main evaluation metric, but still use the Rk statistic for the multiclass problems.

6 Experimental Results

6.1 Sanity Check

Before applying GNNs on the distribution graphs, we want to do a sanity check on generated graphs. We want to prove that a GNN model is able to distinguish different types of graphs. Two types of graphs are randomly generated with the Python package NetworkX [53]. The first class of graphs generated is the Erdős-Rényi graph also known as the binomial graph. The Erdős-Rényi random graph generator returns a graph, $G(n, p)$, where n is the number of nodes and p is the probability of edge creation between nodes. The probability p of creating an edge between two nodes is the same for every edge. In this variant of the Erdős-Rényi graph, a graph has on average $\binom{n}{2}p$ edges and the distribution of the degree for any particular node is binomial:

$$P(\deg(v) = k) = \binom{n-1}{k} p^k (1-p)^{n-1-k} \tag{25}$$

The second class of generated graphs is the power law cluster graph. The graphs are generated using the Holme and Kim algorithm [54], resulting in graphs with power law degree distribution. Power Law degree distribution is a property of the small world network. In these kinds of graphs, a few nodes have many edges while the majority of nodes have a few edges. Many of the graphs in social networks such as Twitter, follow a power law degree distribution. For degree k, k follows a power law, i.e, $p(k)k^{-y}$ where $y > 1$. Specifically, the Holme and Kim algorithm adds m random edges to each node v in graph G, where it adds triangles with a probability of p after each edge creation, i.e. creating an additional edge to one of its neighbors.

100 graphs are generated for each class, where the number of nodes are in the range between 300 and 500 and are evenly distributed for the two classes. The Erdős-Rényi graphs have the probability of edge creation set to $p = 0.5$, while the power law graphs have the number of random edges added to each node to be a random number between $m = 1$ and $m = 5$ for each graph, and the probability of triangle creation to $p = 0.5$.

A Graph isomorphism network (GIN) is trained on the data and is able to classify the graphs with a 99.9% accuracy. As expected, the GNN is able to distinguish the two different classes.

6.2 GIN-Baseline: Dataset 1

For the first classification task we are performing the hyperparameter optimization. The highest scoring configuration is going to be picked for the rest of the classification tasks as well as for Dataset 2. The results of the 8 different configurations can be seen in Table 6 below.

Batch Size	Hidden Units	Layers	Accuracy	Rk-Score
32	32	3	71.9% ± 4.6%	0.455 ± 0.086
32	32	5	64.8% ± 8.1%	0.303 ± 0.157
32	64	2	71.7% ± 4.3%	0.443 ± 0.081
32	64	5	61.3% ± 6.3%	0.228 ± 0.144
128	32	3	**73.8% ± 3.7%**	**0.485 ± 0.060**
128	32	5	71.6% ± 5.2%	0.442 ± 0.106
128	64	2	72.8% ± 3.2%	0.457 ± 0.063
128	64	5	71.1% ± 2.7%	0.431 ± 0.054

Table 6: Hyperparameter optimization on 'AfD' vs 'Grünen'/'Linke'/'CDU' for Dataset 1. The configurations are the same as the ones in the GIN paper [21]. The accuracy and Rk-score seems to be higher for the smaller networks i.e. lower number of hidden units and layers. The bigger batch size, 128, also has better results than the smaller one, 32.

The configuration with the best results is marked with bold text in Table 6. As seen in the Table, there is a lot of variation in performance for the different configurations. For instance, the performance is decreasing when having 5 layers, which might imply that the message propagation in the node embeddings is getting too much spatial information. This is because each layer in the GNN represents the number of hops of information that is being propagated to each node.

After the best hyperparemeter configuration is found, the two remaining classification tasks are trained and evaluated as well. The results of the three different classification tasks using GIN-Baseline are listed below in Table 8. For convenience we list the class-imbalance for the different tasks in Table 7.

	Class-imbalance
AfD vs Grünen/Linke/CDU	53.4%, 46.6%
AfD/CDU vs Grünen/Linke	62.1%, 37.9%
AfD vs CDU vs Grünen vs Linke	53.4%, 8.7%, 16.8%, 21.0%

Table 7: The table shows the class-imbalance for the the three classification tasks, where the 'Class-imbalance' column indicates the percentage of the classes respectively to ordering in the first column.

	Accuracy	Rk-Score
AfD vs Grünen/Linke/CDU	73.8% ± 3.7%	0.485 ± 0.060
AfD/CDU vs Grünen/Linke	69.1% ± 5.6%	0.345 ± 0.116
AfD vs CDU vs Grünen vs Linke	64.8% ± 2.8%	0.433 ± 0.042

Table 8: Results for Dataset 1 for all classification tasks. Because of the class-imbalance (see Table 7) the Rk-Score is used for the main evalutaion metric. The performance drops significantly from the first to second classification task.

As seen in Table 8, the performance goes down when grouping 'AfD' and 'CDU' together. A potential reason for this is that 'AfD' and 'CDU' have significantly different graph structures

which makes them unsuitable to be grouped together. The last classification task, the multiclass problem, even has a higher Rk-Score than the second task. Which means that the model is more capable of classifying the individual parties, compared to when grouped in right ('AfD' & 'CDU') vs left ('Die Grünen' & 'Die Linke') parties (See Section 5.3.1 and Figure 16).

6.3 Baseline: Random Forest Classifier

To compare our GIN-Baseline with a simpler model, we will train a random forest (RF) classifier for all three classification tasks on Dataset 1. The results of the random forest model can be seen in Table 9.

	Accuracy	**Rk-Score**
AfD vs Grünen/Linke/CDU	65.1%	0.297
AfD/CDU vs Grünen/Linke	66.0%	0.260
AfD vs CDU vs Grünen vs Linke	57.1%	0.250

Table 9: Results for random forest classifier on Dataset 1. The results show significantly lower performance for all three classification tasks.

As expected, the RF classifier does not perform as well as the GIN-Baseline. However, it is interesting that the gap in performance between classification task one and two has significantly decreased. The second classification task has almost the same Rk-Score as the first one, and it even has a higher Rk-score compared to the last classification task.

6.4 GNN Comparison Framework

For the comparison framework, all the GNN architectures included in the paper [46] have been run on Dataset 1 except the DGCNN model. The DGCNN model did not execute because of problems with the percentage of the retained nodes in the SortPool layer (see DGCNN paper for details on this [26]). Because of this, DGCNN was excluded from the comparison.

The results of the four models can be seen in Table 10. ECC was the only model that ran out of time and had to be terminated. The time-limit was set to 7 days(168 hours).

	Accuracy	**execution time**
GIN	72.1% ± 3.5%	≈144 hours
DiffPool	67.9% ± 2.9%	≈144 hours
GraphSAGE	66.4% ± 3.7%	≈144 hours
ECC	-	OOR

Table 10: The table shows that GIN is the best performing model among the GNN that executed successfully. It has accuracy that is slightly lower than in Table 8 (differs only by 1.7%). The execution times have been converted from hours to days. Other than the ECC, the models ran for approximately 6 days. OOR means Out of Resources (>168 hours).

As seen in Table 10, the results are not improving from the ones used in our Baseline-GIN. As mentioned before, it is important to point out that our implementation of model selection and model assessment for the GIN-Baseline is different from the comparison framework. Furthermore, the training, testing, and validation splits, in the cross validation are also different. This will have an effect on the overall evaluation of the accuracy of the models and it is therefore not realistic to compare the two results. The purpose of the comparison framework is to

compare the different state-of-the-art GNNs, and Table 10 shows that GIN is indeed the best performing architecture.

We will assume that the change in data from Dataset 1 to Dataset 2 will not have a substantial impact on the difference in performance between the GNN models in the comparison framework. We will therefore move forward with the GIN-baseline model, and exclude the comparison framework from any further experiments.

6.5 Dataset 2

The results of experiments on Dataset 2 with the GIN-baseline is listed below.

	Accuracy	**Rk-Score**
AfD vs CDU/Grünen	73.7% ± 1.8%	0.481 ± 0.034
AfD/CDU vs Grünen	77.4% ± 1.4%	0.556 ± 0.026
AfD vs CDU vs Grünen	66.5% ± 1.7%	0.502 ± 0.024

Table 11: Results of experiments on Dataset 2. As described in Section 5.3.4 and visualized in Figure 17 the data is resampled to balanced sets in each classification task. The accuracy metric is therefore a valid metric to use for the binary classification task, but we will still keep the Rk-Score for comparison with experiments on Dataset 1 and for the multiclass problem. Compared to Table 8, the ranking of performance among the classification tasks have changed with the second classification task having highest performance.

The results in Table 11 shows better performance than in Table 8. However, since we unfortunately had to exclude Die Linke from experiments on Dataset 2, the experiments are not the same for Dataset 1 and Dataset 2. It is therefore difficult to compare the two. However, it is still worth pointing out some differences in the results in Table 11 compared to Table 8. Our assumption that combining 'Die Linke' and 'CDU' resulted in a weaker performance in Table 8 because of the structures being different in the two parties does not seem to hold for Table 11, where the performance increase when combining the two parties. This could be due to the absence of Die Linke in Dataset 2.

Table 11 shows that the performance does not decrease when we have more sources and more data. Furthermore, as mentioned in Section 4.5, Dataset 2 has a lower average of tweets per user than Dataset 1, which means that we are giving a large number of sources the same label, i.e. fewer data points per user compared to Dataset 1.

As a proof of concept of the relationship between number of sources in the grouping and the performance of the model, we do a simple experiment where we only have two politicians from two different parties that we wish to classify. For this experiment we pick a politician from 'Die Linke' and one from 'Die Grünen' where we have the same amount of graphs. Since the dataset is balanced, we are only using accuracy as a evaluation metric.

	Accuracy
Malte Kaufmann (AfD) vs Goering Eckardt (Die Grünen)	89%

Table 12: Results of classifying two individual politicians.

As expected, we get a much higher accuracy score on this simple dataset with only two sources. This proves that, at least for this example, the GNN is able to distinguish individual sources with a high accuracy.

6.6 Dataset 3

Since the data are completely different in Dataset 3, we are again doing a hyperparameter optimization. The optimization is done on the main classification task of interest namely '5G Conspiracy' vs 'Non Conspiracy' with timestamps as node features. The results of the hyperparameter optimization are listed in Table 13.

Batch Size	**Hidden Units**	**Layers**	**Accuracy**	**Rk**
32	32	3	64.2% ± 8.4%	0.287 ± 0.169
32	32	5	62.3% ± 8.0%	0.258 ± 0.161
32	64	2	64.8% ± 7.2%	0.299 ± 0.144
32	64	5	64.8% ± 8.9%	0.301 ± 0.179
128	32	3	**67.4% ± 9.0%**	**0.354 ± 0.178**
128	32	5	62.7% ± 7.4%	0.263 ± 0.149
128	64	2	64.4% ± 6.7%	0.295 ± 0.133
128	64	5	66.7% ± 7.9%	0.342 ± 0.158

Table 13: Hyperparameter Optimization on '5G Conspiracy' vs 'Non Conspiracy' for Dataset 3 with timestamps. The same configuration as in Table 6 results in the best performance for Dataset 3. Altough, the variation in performance between the configuration is much smaller than in Table 6.

From Table 13 we can see that the same hyperparamet configuration as in Dataset 1 has the highest performance among the configurations. We pick this as our optimal configuration and use it on the rest of the classification tasks. The results of all three classification tasks for Dataset 3 can be seen in Table 14 below.

	Accuracy	**Rk-Score**
5G Conspiracy/Conspiracy vs Non Conspiracy	62.4% ± 2.4%	0.252 ± 0.048
5G Conspiracy vs Non Conspiracy	67.4% ± 9.0%	0.354 ± 0.178
Conspiracy vs Non Conspiracy	62.2% ± 5.6%	0.249 ± 0.112

Table 14: Results of experiments on Dataset 3 with timestamps. The best performing classification taks is '5G Conspiracy' vs 'Non Conspiracy', and the two other classification tasks have almost identical performance.

The performance is lower than the results of the experiments on the politician datasets. However, the data is entirely different in Dataset 3, thus expecting similar results as in Dataset 1 and Dataset 2 does not make sense.

We can see that '5G Conspiracy' vs 'Non Conspiracy' has a significantly better performance in Table 14. This is to be expected, since the 'Conspiracy' class contains a much bigger range of tweets, making the task more challenging.

To evaluate the importance of the timestamps as features, we will conduct the same experiments on Dataset 3 without the timestamps. The results can be seen in Table 15 below.

Table 15 shows that the accuracy drops for all three experiments, but it only drops significantly for '5G Conspiracy' vs 'Non Conspiracy'.

As a last experiment, Dataset 3 is randomly relabeled into two equal sized arbitrary classes. This is done to see if our approach is viable. When the data is randomly shuffled and cat-

	Accuracy	Rk-Score
5G Conspiracy/Conspiracy vs Non Conspiracy	60.4% ± 3.7%	0.236 ± 0.1.06
5G Conspiracy vs Non Conspiracy	62.3% ± 9.0%	0.255 ± 0.187
Conspiracy vs Non Conspiracy	61.5% ± 5.5%	0.236 ± 0.106

Table 15: Results of experiments on Dataset 3 without timestamps. The overall performance decreases, but it is only for '5G Conspiracy' vs 'Non Conspiracy' that it decreases significantly.

egorised into two classes, the GNN should not be able to pick up a signal in the data. The results can be seen in Table 16 below.

	Accuracy
Randomly relabeled Dataset 3	50.3% ± 0.8%

Table 16: Results of classifying a randomly relabeled Dataset 3 of two equal sized classes. It has almost 50% accurcay, meaning that the model is not able to pick up a signal in the data.

The results in Table 16 are as expected, i.e. no better than random guessing. The GNN is not able to learn anything meaningful from the data when the data is randomly assigned to two classes. This supports the reliability of our results in Table 14 and 15.

7 Discussion

This chapter will discuss the results presented in Chapter 6. We intend to give an intuition behind the results and compare the different experiments for the different datasets. In the end we will discuss some of the limitations of our approach and give suggestions for future work.

7.1 Graph Structure

The sanity check in Section 6.1 proves that if there is a huge difference in graph structure, the GNN is able to distinguish the graphs. The signal in our data, i.e. the pattern in the data that lets us generalize the model to new data, is not always strong. There is no criteria that says that graphs related to different political parties have to look differently, or that graphs related to fake news have to look differently than graphs related to real news. Because of this, it is difficult to have an indication of the accuracy one can expect from the experiments. On the other hand, we know from previous work that there are characteristics that are different from how fake news propagates compared to real news [2]. Since our experiments on Dataset 3 show significantly better results than random guessing, we can say with certainty that there is something in the structure that separates the classes. The last experiments in Table 16 further proves this, since it performs as bad as random guessing when the Dataset 3 is randomly relabeled into two arbitrary classes.

7.2 Network Size

In Table 6 and 13, we see that the smaller networks work better than the larger ones, i.e. number of hidden units and number of layers. This might mean that the larger neural networks are overfitting on the training data.

For the number of hidden layers, which is the number of hops of neighborhood information we are aggregating in the graphs, if the number is larger than the diameter of a graph, we are essentially aggregating node information from the whole graph for each node embedding. Aggregating information from the whole graph is not very useful, since all node embeddings will essentially receive the same information.

From the diameter plots in Section 4 (Figure 8, 11, 15), most of the graphs do not exceed a diameter of 5 (especially for Dataset 1 and Dataset 2). Furthermore, when using node degree as node features, we already have information from the first layer, since node degree contains information of a node's immediate neighbors. It therefore makes sense that the shallower networks perform better on the data. If the data would have had more variation in size and connectivity, adaptive depth could have been a preferable solution. Adaptive depth adjusts the number of layers based on the connectivity of the graphs, meaning that shallower networks are used on highly connected graphs to avoid including too much neighborhood information in the node embeddings, and deeper networks are used on less connected graphs to include sufficient neighborhood information.

7.3 GNN vs Baseline

Table 9 shows that the Random Forest Classifier has a lower performance than our GIN-Baseline. This proves that using a classifier with manual feature extraction is less preferable than using a GNN. However, a more sophisticated feature extraction could potentially improve the performance of the RF classifier. On the other hand, that is also the main advantage of using GNNs, that learning relevant graph features is part of the training process.

7.4 Comparison of GNNs

The result of the comparison framework on Dataset 1 (Table 10), proves that GIN is indeed the best performing model among the GNNs. This is not unexpected, since it has the best performance on the social datasets among the GNNs in the framework in Ericca et al [46]. That is also the reason why we chose GIN as the initial architecture for our model, and Table 10 further proves that this is the best choice for our particular dataset.

GIN, GraphSAGE and DiffPool ran in parallel for approximately 6 days. The long execution time is due to the hyperparameter optimization in the cross-validation, which uses between 32 and 72 different configurations. The execution time would have greatly increased if more data was provided, as in Dataset 2. This made Dataset 1 ideal for benchmarking the different GNNs. DGCNN was not implemented because it required additional modification of the framework and ECC ran out of resources after running for 7 days, which was set to be the time limit. However, based on the results in Ericca et al. [46] it is unlikely that DGCNN and ECC would have performed better than GIN. The only GNN architecture that performed better than GIN on any of the datasets in Ericca et al [46] was GraphSAGE.

7.5 Dataset Comparison

Table 10 shows that performance is approximately the same for Dataset 2 compared to Dataset 1. We have to be careful when comparing the results of Dataset 1 and Dataset 2, since 'Die Linke' is excluded from Dataset 2. Furthermore, the results of Dataset 1 are also on unbalanced sets, meaning that the accuracy scores in Table 7 are not as meaningful for some of the experiments. However, the Rk-Score takes the class-balance into account and can therefore be compared across balanced and unbalanced sets. Even though Figure 10 shows that most of the graphs in Dataset 2 are coming from a few sources compared to the overall number of sources, the result of Dataset 2 still proves that the performance does not decline when increasing the number of individual sources and data.

Experiments on Dataset 1 and Dataset 2 have better results than experiments on Dataset 3. However, it does not make much sense to compare the two cases, since the data is completely different. For Dataset 1 and Dataset 2 we have data that is labeled by source, whereas Dataset 3 is labeled by the content. Table 11 proves that we are able to distinguish individual sources with a relatively high accuracy. This is something that only works for Dataset 1 and Dataset 2, since Dataset 3 can not rely on sources. In other words, the two cases are two different problems. We also have less data in Dataset 3 compared to the other two, but on the other hand we have larger graphs in Dataset 3. The variation in data size is also what makes the models have different standard deviation in the cross-validation. We can see in Table 11 that experiments on Dataset 2 have low standard deviations, i.e. little variation in the evaluation of the folds, while Table 15 shows much higher values of standard deviations for experiments on Dataset 3.

7.6 Future Work

Since the data in Dataset 3 differ from Dataset 1, it could be interesting to run the comparison framework on Dataset 3. However, due to time limitations this was not prioritized. Furthermore, based on the results we got in Table 10 and the results in Ericca et al [46], it is probable to assume that GIN would perform the best.

In all of our datasets we do not have particularly large graphs. We are assuming that the

accuracy will go up when graphs become bigger, since they contain more information. In real life scenarios, we want to analyze news that have spread far and wide. Therefore in practice we would work with much larger graphs. However, it is a difficult process to obtain these graphs since they are not just spreading on one domain such as Twitter, but they are spreading across the internet over different domains. Furthermore, a user does not have to retweet a tweet to be able to read it or share it with friends and people on the internet.

Another limitation to our model is that we have to convert the timestamps to labels. It would obviously be beneficial to keep the timestamp as a continuous variable in order to not loose information when converting it.

Combining the GNN model with a content-based approach such as NLP would potentially result in a much more robust model. However, this would require text based data additional to the graph data we have been working on.

8 Conclusion

The objective of this thesis was to classify distribution graphs from Twitter with Graph Neural Networks (GNNs), with the main agenda of applying the methods to identify fake news. Previously, automatic fake news detection has been tackled with Machine Learning (ML) methods that typically have been limited to work on Euclidean data and feature engineering. In many scenarios where one has complex relations within the data, a graph structured representation of the data is more suitable. Because of its complexity and arbitrary size, graph structured data can not be applied to these traditional machine learning methods without some kind of preprocessing or manual feature extraction.

GNNs are extensions to neural networks that allow modelling on graph structured data of arbitrary size and structure. The GNNs can capture the complex relationships and dependencies in the data, and do not rely on manual feature extraction. We have applied GNNs on distribution graphs collected from Twitter, where the graphs represent the propagation of tweets on the platform. Specifically, we have been working with three different datasets, where the first two datasets contained tweets from German politicians and the third datasets contained tweets related to the Covid-19 pandemic where tweets have been manually labeled as fake news and non-fake news. Modelling with GNNs on the German politician datasets showed promising results for classification of different political parties, resulting in an Rk-Score of 0.485 (accuracy of 73.8%) for 'AfD' vs 'Grünen/Linke/CDU'. Furthermore, it proved to be more accurate than a simple baseline model that relied on feature engineering and had only an Rk-Score of 0.297 (accuracy of 65.1%) for the same classification task.

We performed a fair comparison of state-of-the-art GNNs based on the framework proposed in Errica et al. [46]. It showed that the Graph Isomorphism Network (GIN) indeed has the best performance among the GNNs. With 72.1% accuracy on the 'AfD' vs 'Grünen/Linke/CDU' experiment, GIN performed 4.2% better than DiffPool and 5.7% better than GraphSAGE, while ECC terminated since it ran out of resources after 7 days of training.

The performance of the models dropped slightly when applying the same methods to the fake news dataset, having a Rk-Score of 0.354 (accuracy of 67.4%) for '5G Conspiracy' vs 'Non Conspiracy'. However, in the fake news dataset we have a much larger number of individual sources, meaning the model can not rely on characteristics from individual sources. The Rk-Score dropped further to 0.255 (accuracy of 62.3%) when excluding the timestamps as node fatures, showing that the timestamps do contain meaningful information. We further proved that our prediction accuracy is not due to statistical randomness or noise in the data by randomly assigning labels of the fake news dataset and classifying it, resulting in 50.3% accuracy. Even though our accuracy is not very high for the fake news dataset, the result still proves that structure based fake news classification is possible.

References

[1] "Bbc," https://www.bbc.com/news/newsbeat-52395771.

[2] Soroush Vosoughi, Deb Roy, and Sinan Aral, "The spread of true and false news online," *Science*, vol. 359, no. 6380, pp. 1146–1151, 2018.

[3] "FakeNews: Corona virus and 5G conspiracy task at The MediaEval Multimedia Evaluation benchmark 2020," `https://multimediaeval.github.io/editions/2020/tasks/fakenews/`.

[4] Y. LeCun, P. Haffner, L. Bottou, and Y. Bengio, "Object recognition with gradient-based learning," *Shape, Contour and Grouping in Computer Vision*, pp. 319–45, 319–345, 1999.

[5] JL ELMAN, "Finding structure in time," *Cognitive Science*, vol. 14, no. 2, pp. 179–211, 1990.

[6] Matthias Fey and Jan Eric Lenssen, "Fast graph representation learning with pytorch geometric," 2019.

[7] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., pp. 8024–8035. Curran Associates, Inc., 2019.

[8] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and MIT Press, *Deep learning*, The MIT Press, 2016.

[9] Diederik P. Kingma and Jimmy Lei Ba, "Adam: A method for stochastic optimization," *3rd International Conference on Learning Representations, Iclr 2015 - Conference Track Proceedings*, 2015.

[10] A. Lehman B. Weisfeiler, "Reduction of a graph to canonical form and algebra generated by process," *Nauchno-tekhnicheskaya Informatsiya Seriya 2-informatsionnye Protsessy I Sistemy*, , no. 9, pp. 12–, 1968.

[11] Alessandro Sperduti and Antonina Starita, "Supervised neural networks for the classification of structures," *Ieee Transactions on Neural Networks*, vol. 8, no. 3, pp. 714–735, 1997.

[12] M Gori, G Monfardini, and F Scarselli, "A new model for learning in graph domains," *Proceedings of the International Joint Conference on Neural Networks (ijcnn), Vols 1-5*, pp. 729–734, 2005.

[13] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini, "The graph neural network model," *Ieee Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, 2009.

[14] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu, "A comprehensive survey on graph neural networks [arxiv]," *Arxiv*, pp. 22 pp., 22 pp., 2019.

[15] Zhiyuan Liu and Jie Zhou, "Introduction to graph neural networks," *Synthesis Lectures on Artificial Intelligence and Machine Learning*, vol. 14, no. 2, pp. 1–127, 2020.

[16] W. A. Kirk, Mohamed A. Khamsi, and Inc ebrary, *An introduction to metric spaces and fixed point theory*, John Wiley, 2001.

[17] Jure Leskovec, "Cs224w: Machine learning with graphs: Lecture 8 graph neural networks," Stanford University Lecture, 2020.

[18] William L. Hamilton, Rex Ying, and Jure Leskovec, "Inductive representation learning on large graphs," 2018.

[19] Thomas N. Kipf and Max Welling, "Semi-supervised classification with graph convolutional networks [arxiv]," *Arxiv*, pp. 10 pp., 10 pp., 2016.

[20] S Hochreiter and J Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–80, 1735–1780, 1997.

[21] Keyulu Xu, Stefanie Jegelka, Weihua Hu, and Jure Leskovec, "How powerful are graph neural networks?," *7th International Conference on Learning Representations, Iclr 2019*, 2019.

[22] K HORNIK, "Approximation capabilities of multilayer feedforward networks," *Neural Networks*, vol. 4, no. 2, pp. 251–257, 1991.

[23] Keyulu Xu, Chengtao Li, Yonglong Tian, Tomohiro Sonobe, Ken Ichi Kawarabayashi, and Stefanie Jegelka, "Representation learning on graphs with jumping knowledge networks," *35th International Conference on Machine Learning, Icml 2018*, vol. 12, pp. 8676–8685, 2018.

[24] Martin Simonovsky and Nikos Komodakis, "Dynamic edge-conditioned filters in convolutional neural networks on graphs [arxiv]," *Arxiv*, pp. 12 pp., 12 pp., 2017.

[25] Rex Ying, Jiaxuan You, Christopher Morris, Xiang Ren, William L. Hamilton, and Jure Leskovec, "Hierarchical graph representation learning with differentiable pooling," *Advances in Neural Information Processing Systems 31 (nips 2018)*, vol. 31, 2018.

[26] Muhan Zhang, Zhicheng Cui, Marion Neumann, and Yixin Chen, "An end-to-end deep learning architecture for graph classification," *32nd Aaai Conference on Artificial Intelligence, Aaai 2018*, pp. 4438–4445, 2018.

[27] Jianxin Ma, Xiao Wang, Peng Cui, and Wenwu Zhu, "Hierarchical taxonomy aware network embedding," *Proceedings of the Acm Sigkdd International Conference on Knowledge Discovery and Data Mining*, pp. 1920–1929, 2018.

[28] Xinyi Zhou and Reza Zafarani, "A survey of fake news: Fundamental theories, detection methods, and opportunities," *Acm Computing Surveys*, p. 37, 2020.

[29] "Politifact," `https://www.politifact.com/`.

[30] "Factcheck," `https://www.factcheck.org/`.

[31] "Gossipcop," `https://www.gossipcop.com/`.

[32] "Hoax-slayer," `https://www.hoax-slayer.net/`.

[33] "Fiskkit," ://fiskkit.com/.

[34] Xinyi Zhou, Atishay Jain, Vir V. Phoha, and Reza Zafarani, "Fake news early detection: A theory-driven model [arxiv]," *Arxiv*, pp. 24 pp., 24 pp., 2019.

[35] Song Feng, Ritwik Banerjee, and Yejin Choi, "Syntactic stylometry for deception detection," *50th Annual Meeting of the Association for Computational Linguistics, Acl 2012 - Proceedings of the Conference*, vol. 2, pp. 171–175, 2012.

[36] Verónica Pérez-Rosas, Bennett Kleinberg, Alexandra Lefevre, and Rada Mihalcea, "Automatic detection of fake news," 2017.

[37] Tianqi Chen and Carlos Guestrin, "Xgboost: A scalable tree boosting system," *Proceedings of the Acm Sigkdd International Conference on Knowledge Discovery and Data Mining*, vol. 13-17-, pp. 785–794, 2016.

[38] Yoon Kim, "Convolutional neural networks for sentence classification," 2014.

[39] Carlos Castillo, Marcelo Mendoza, and Barbara Poblete, "Information credibility on twitter," *Proceedings of the 20th International Conference Companion on World Wide Web, Www 2011*, pp. 675–684, 2011.

[40] Sejeong Kwon, Meeyoung Cha, Kyomin Jung, Wei Chen, and Yajun Wang, "Prominent features of rumor propagation in online social media," *Proceedings - Ieee International Conference on Data Mining, Icdm*, pp. 6729605, 1103–1108, 2013.

[41] Jing Ma, Wei Gao, and Kam Fai Wong, "Rumor detection on twitter with tree-structured recursive neural networks," *Acl 2018 - 56th Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference (long Papers)*, vol. 1, pp. 1980–1989, 2018.

[42] Qi Huang, Chuan Zhou, Jia Wu, Mingwen Wang, and Bin Wang, "Deep structure learning for rumor detection on twitter," *Proceedings of the International Joint Conference on Neural Networks*, vol. 2019-, pp. 8852468, 2019.

[43] Jing Ma, Wei Gao, and Kam Fai Wong, "Rumor detection on twitter with tree-structured recursive neural networks," *Acl 2018 - 56th Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference (long Papers)*, vol. 1, pp. 1980–1989, 2018.

[44] Daniel Thilo Schroeder, Konstantin Pogorelov, and Johannes Langguth, "Fact: A framework for analysis and capture of twitter graphs," *2019 6th International Conference on Social Networks Analysis, Management and Security, Snams 2019*, pp. 8931870, 134–141, 2019.

[45] Kristian Kersting, Nils M. Kriege, Christopher Morris, Petra Mutzel, and Marion Neumann, "Benchmark data sets for graph kernels," 2016.

[46] Federico Errica, Marco Podda, Davide Bacciu, and Alessio Micheli, "A fair comparison of graph neural networks for graph classification," 2020.

[47] Leo Breiman and E. Schapire, "Random forests," 2009.

[48] Google Research., "Google colaboratory," `https://colab.research.google.com`.

[49] Simula Research Laboratory., "ex3 simula," `https://www.ex3.simula.no/`.

[50] Deutscher Bundestag, "Sitzverteilung im 19. deutschen bundestag," https://www.bundestag.de/parlament/plenum/sitzverteilung_19wp.

[51] J. Gorodkin, "Comparing two k-category assignments by a k-category correlation coefficient," *Computational Biology and Chemistry*, vol. 28, no. 5-6, pp. 367–374, 2004.

[52] B W Matthews, "Comparison of the predicted and observed secondary structure of t4 phage lysozyme," *Biochimica Et Biophysica Acta*, vol. 405, no. 2, pp. 442–51, 442–451, 1975.

[53] Daniel A. Schult, "Exploring network structure, dynamics, and function using networkx," 2015.

[54] Petter Holme and Beom Jun Kim, "Growing scale-free networks with tunable clustering," *Physical Review. E, Statistical, Nonlinear, and Soft Matter Physics*, vol. 65, no. 2, pp. 026107, 026107, 2002.