Structural Anomaly Detection in Knowledge Graphs Using Graph Neural Networks

Henrik Syversen Johansen

Thesis submitted for the degree of
Master in Informatics: Programming and System Architecture
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Autumn 2022

Structural Anomaly Detection in Knowledge Graphs Using Graph Neural Networks

Henrik Syversen Johansen

Structural Anomaly Detection in Knowledge Graphs Using Graph Neural Networks

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

Anomalies are in general rare or somehow different elements among a set of elements. Identifying such elements has several important applications within the realms of financial fraud, review fraud, health care, security and others. In this work we propose a definition of what it means for a substructure of a knowledge graph to be anomalous. We also create AI systems that use end-to-end graph neural networks to classify these substructures as anomalous or not. In evaluating the performance of these AI systems we show that while they are much faster than a precise symbolic implementation of the definition of anomalous substructures, the results are expectedly lower with regards to classification performance. In real-life scenarios where this loss in classification performance is tolerable however, we have shown that our approach is feasible.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my primary supervisors Egor V. Kostylev and Basil Ell for providing me with invaluable input and guidance throughout this project. I would also like to thank my family and friends for their unconditional support and camaraderie

Chapter 1

Introduction

The field of anomaly detection has numerous important real-life applications of societal value. Including fraud detection in the context of e-commerce and other financial activity [5], review fraud [27], detection of bots and fake social media accounts [8], health insurance fraud [19], and network security [13].

Most previous research on anomaly detection focuses on finding anomalous entities, for example Twitter accounts with an abnormal posting frequency, or a financial account with an abnormal volume of trades. These approaches rely on some combination of the features of an entity to mark it as anomalous or not [7].

However, entities that are not considered anomalous by their features alone, may yet be anomalous in their relation to other entities. An example may be a group of Twitter accounts that individually have a normal posting frequency, and post human looking Tweets, but appear to be coordinated due to similar post timings and Tweet content. Another example may be a set of e-commerce accounts that sell widely different products, yet all seem to review each other's accounts as trustworthy without ever posting reviews on other accounts.

When entities irrespective of their connection to other entities are represented in data structures, they are often done so in a tabular format such as CSV, spreadsheet tables or in a relational database. While tabular data formats can, in principle represent the connections between entities, knowledge graph formats such as RDF do so in a more direct manner, with entities represented entirely in their connections to other entities [32].

Existing approaches to anomaly detection have both been of symbolic and non-symbolic nature. The former employs expert knowledge and logical reasoning to identify anomalies in some mathetmatical way, while the latter is based on statistical methods often from the domain of AI and machine learning [7].

In the context of anomalies in graph structured data, symbolic approaches have been used [1]. However due to the high complexity and size of real-life knowledge graphs, the symbolic approaches are often infeasibly slow. As progress have been made on neural network architectures that process graph structured data direcly [3], the potential

for non-symbolic solutions to the problem of anomaly detection in graphs appears.

These graph neural networks have been used for the purposes of anomaly detection in graphs. However, the anomalies that the aproaches focus on are often either "anomalous nodes" or "anomalous edges" [33]. In the case where they attempt to identify anomalous structures, it is often in the context of anomalous *changes* in structures from one time slice of a dynamic graph to the next [6].

Furthermore, several of the existing approaches do not employ graph neural networks in an end-to-end manner. They often use graph neural networks to map the graph structured data to an embedding space on which the classification step is performed using traditional approaches such as support vector machines [33] or RNNs [6, 34].

In this thesis we provide a precise, readily implementable, symbolic definition of what anomalous substructures of a graph are. We also present a SPARQL based implementation of this symbolic definition.

We also design several AI systems that learn to classify anomalous substructures as we have defined them. These systems all have in common that they consist of an embedding and aggregation component, while they differ in their combinations of particular embedding and aggregation components. The embedding component embeds the nodes of a graph using either GraphSage [16], a GCN [18], or a RGCN [29]; while the aggregation component aggregates the embeddings of a the set of nodes that define our substructure, either using a simple mean or a feed forward neural network.

We implement these systems using the PyTorch deep learning library, and train and evaluate them the benchmarks created using the above mentioned SPARQL based implementation of the symbolic definition of stuctural anomalies.

Our evaluation results show that these AI systems provide a considerable increase in execution speed compared to the precise SPARQL implementation. However, as is expected of AI systems they loose out in terms of classification performance, although this loss may be tolerable in real-life situations when speed is the most important factor.

Future work may explore training and evaluation of our suggested AI systems on benchmarks based on more complex graphs and with larger substructures that we were not able to explore due to computational constraints. It may also take aim at introducing some normalization factor to our definition of structural anomalies so that patterns considered anomalous or not in subgraphs of larger graphs, are more likely to be considered anomalous or not in the larger graph as well.

Part I

Background

Chapter 2

Knowledge Graphs

Traditionally most data is represented in large tabular databases. Either in file formats such as *csv* or in relational databases. In tabular databases columns correspond to some aspect of the data being represented, and rows correspond to a concrete instance of what is being represented.

Knowledge graphs on the other hand represent data in a graph structure consisting of nodes and edges connecting those nodes. Knowledge graphs are well suited for representing data where the connections between pieces of data are intrinsically informative. While table based data representations can also represent the connections between pieces of data, knowledge graphs do so in a more direct way.

Knowledge graphs may only represent elements and their connections and thus only structural information. In this case all nodes are essentially the same, and all edges are the same. The only information stored in the knowledge graph is which node is connected to which node. However, graphs may also be *attributed* or *colored* in which case nodes and edges may both have attributes assigned to them.

2.1 RDF

The Resource Description Framework (RDF) is a framwork developed by the World Wide Web Consortium (W3C) [32] with the intention of making information on the internet machine readable. In RDF information is stored as *triples*. A *triple* consist of a *subject* a *predicate* and an *object*. For example, to describe the fact that Oslo is the capital of Norway, the following triple could be used: *Oslo - Capital Of - Norway*.

2.1.1 IRIs

Internationalized Resource Identifier (IRIs) are resource identifiers that on an international scale uniquely identify any RDF resource. An IRI is thus essentially the globally unique name of any single subject, object or predicate in an RDF graph.

2.1.2 Format for RDF Storage

While RDF is a common framework for representing relational data in the form of subject, predicate, object triples. The syntax through which RDF information is serialized and stored may vary. Three commonly used formats are RDF/XML, Turtle, and JSON-LD.

RDF/XML was introduced alongside W3C RDF specification, and was thus the first serialization format used. However, due to being more compact readable by humans Turtle is now often used in practice. JSON-LD is built on JSON and allows RDF serialization in a JSON like format. Here is the same triple represented using the three formats:

XML

```
<?xml version="1.0" encoding="utf-8" ?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:ns0="http://example.org/">

  <rdf:Description rdf:about="http://example.org/oslo">
    <ns0:capital-of rdf:resource="http://example.org/Norway"/>
  </rdf:Description>

</rdf:RDF>
```

Turtle

```
@prefix ns0: <http://example.org/> .

ns0:oslo ns0:capital-of ns0:Norway .
```

JSON-LD

```
[
    {"@id":"http://example.org/Norway"},
    {"@id":"http://example.org/oslo","http://example.org/capital-of":[
        {"@id":"http://example.org/Norway"}]
    }
]
```

2.2 SPARQL

SPARQL is a query language designed to query RDF graphs. It's syntax is somewhat similar to SQL, which is used to query tabular data, with some key differences. In this thesis we will be using the "SELECT WHERE" SPARQL query. While the "SELECT WHERE" SQL query returns a set of columns from a table, with it's rows filtered by some condition. The "SELECT WHERE" SPARQL query returns a mapping from variables to IRIs (i.e. node identifiers), filtered by a set of conditions expressed as "subject predicate object" triples. As such the SQL query returns rows in a table, while the SPARQL query returns IRIs that match a pattern / set of triples.

Below is an example of a an SQL query that queries a table with the following columns: “country”, “capital”, and “continent”, for countries in Europe and their capital cities:

```
SELECT country, capital FROM countries
WHERE continent = "Europe";
```

The result of the query is a table with the columns: “country” and “capital” and each row representing a given country with it’s capital.

Given an RDF graph consisting of the following two triples for all countires:

```
@prefix : <http://example.org/> .

:country :hasCapital :capital .
:country :inContinent :continent .
```

Where :country, :capital and :continent are variables for each country its capital and continent.

A SPARQL query for each country in Europe and its capital city is the following:

```
PREFIX : <http://example.org/>

SELECT ?country ?capital WHERE {
    ?country :inContinent :Europe
}
```

The results are returned as a list of sets of IRIs that map to the variables “?country” and “?capital”.

Chapter 3

Graph Datasets

3.1 The Lehigh University Benchmark (LUBM)

The Lehigh University Benchmark (LUBM) is a benchmark intended to evalute the performance of various Semantic Web repositories using a standardized generated ontology along with a set of defined queries [21]. In this thesis we use the ontology generator from LUBM to create synthetic graphs of an arbitrary size that we look for anomalies in.

The LUBM graph generation procedure returns a ontology as a RDF graph that conceptually represents a set of universities with departments, it's courses, students, and employees.

3.2 IMDB

The IMDB dataset is a graph consisting of a subset of the online movie database www.IMDB.com [15]. It's nodes represent movies, directors and actors. The size of the graph as well as the number of node and edge types are shown in table 3.1.

3.3 LastFM

The LastFM dataset is a graph based on information from the website www.last.fm. The website tracks artists, their songs and which songs and artists users follow. The dataset consists of nodes representing users, artists, and artist tags [15]. Again, the size of the graph as well as the number of node and edge types are shown in table 3.1.

Dataset	# Nodes	# Edges	# Node Types	# Edge Types
IMDB	11616	34212	3	4
LastFM	20612	201908	3	5
LUBM	Variable	Variable	14	12

Table 3.1: The number of nodes and edges, as well as node and edge types in our graphs

Chapter 4

Anomaly Detection

4.1 In Graphs

Graph anomaly detection is a problem that can be hard to define in a general and still useful manner. However a survey by Akoglu, Tong and Koutra [1] defines the problem of graph anomaly detection as such:

> Given a (plain/attributed, static/dynamic) graph database, find the graph objects (nodes/edges/substructures) that are rare and differ significantly from the majority of the reference objects in the graph.

This definition is general and somewhat uninformative without context. Also, what is meant by *plain*, *attributed*, *static* and *dynamic*, is not immeditately clear. As such those terms are worth going into. It is also unclear how to interpret rarity in a quantifiable manner.

4.1.1 Static, Dynamic, Plain or Attributed Graphs

A *static* graph is simply one that does not change. No nodes are modified, and no edges are modified. *Dynamic* graphs on the other hand, change over time. An example of a static graph may be a snapshot of a RDF database, while an example of a dynamic one might be the changes in that RDF database over time.

Both static and dynamic graphs may be *plain* or *attributed*. Plain graphs consist of nodes and the edges between them, that's it. All the information about the graph is structural, and no node is different from another when compared one-on-one. In attributed graphs however, nodes or edges may have attributes associated with them.

An example of plain graphs are vanilla RDF graphs, with OWL data properties however, nodes may be attributed. Edge attributes may also be emulated by adding nodes with data properties representing the attributes in between the nodes for every object property.

4.1.2 The Intepretation of Rarity

The concrete definition of rarity varies between problem settings. Nevertheless, something that is rare usually has a rare combination of categorical values, numerical values, or graph structural characteristics (e.g. the degree of a node, or the average degree of nodes in a substructure).

4.1.3 Concrete Formalizations and Approaches

The following concrete formalizations of anomaly detection is from the survey by Akoglu, Tong and Koutra [1].

In Static Graphs

> Given the snapshot of a plain or attributed graph database, find the nodes and/or edges and/or substructures that are "few and different" or deviate significantly from qthe patterns observed in the graph.

The main approaches for detecting anomalies in static graphs mentioned in the survey is as follows: Structure based methods and community based methods.

Structure based methods focus on the structural information of the graph, as well as attributes of the structural elements in the case of attributed graphs.

Community based methods rely on identifying clusters of connected nodes and classifying the elements that bridge these clusters, or that are not part of the clusters as anomalous. In the case of attributed graphs the outlierness of an elements attributes within a community is also considered as anomalous.

In Dynamic Graphs

> Given a sequence of plain or attributed graphs, find time timesteps that corresponds to a change or event, as well as the top-k nodes, edges, or parts of the graphs that contribute most to the change.

For anomaly, or event -detection in dynamic graphs hte survey identifies four main categories of approaches Akoglu, Tong and Koutra [1]. Feature based event detection, decomposition based event detection, community based event detection, and window based event detection.

Feature based event detection methods compare summaries of graph features (e.g.) at various time slices in order to identify anomalous time slices based on their dissimilarity to the other time slices.

Decomposition based event detection methods employ matrix or tensor decompositions and evaluate the resulting eigenvectors, eigenvalues and singular values.

Community based event detection methods focus on graph communities or clusters and their structural and contextual changes over time. Events are tracked when changes occur.

Window based event detection methods judge normal "behavior" in a graph over a time window, when a time slice deviates from this behavior it's marked as anomalous.

Chapter 5

Machine Learning

Machine learning is an approach to solving problems indirectly by creating algorithms that "learn" an approximation of a function rather than a human having to design a function that solves the problem explicitly.

There are broadly two categories of machine learning approaches: supervised and unsupervised learning. In the former case the system is provided both inputs and expected outputs, and learns by attempting to match it's outputs to the expected outputs. In the latter case, the system is provided no expected outputs.

Examples of supervised machine learning approaches are: Naive Bayes classifiers, an approach based on using Bayes' theorem with an assumption of independence among the features of the elements; Decision tree learning, an approach that learns which features partition the dataset in the most informative way (as defined by a reduction in information entropy); and Support Vector Machines (SVM) that attempt to separate classes with the largest margin in some n-dimensional space.

Examples of unsupervised machine learning approaches include: K-means clustering that attempt to identify clusters of elements in some feature space; and DBSCAN that attempts to group elements that are closely packed in some feature space.

5.1 Deep Learning

A perceptron is an interesting supervised machine learning approch inspired by the neurons of the brain. It takes a vector of features as an input, computes the dot product of the feature vector with some weight matrix, adds a bias variable to the result, and outputs the result. In stacking multiple layers of these perceptrons you can create deep networks of perceptrons. These deep networks are known as deep neural networks, or as feed forward networks.

While the term deep learning initially referred to these deep neural networks of stacked perceptron layers, it has grown to encompass other derivative methods, often with approaches very different from the initial multilayer perceptron.

5.2 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) process sequences of feature vectors, reusing it's weights for each feature vector in the sequence [10]. At each point in the sequence the processed output of the previous element in the sequence is processed alongside the current element. In that way this approach can process sequential data in a way where information about the order of elements is preserved. As it is iterative it can also process sequences of arbitrary length.

Derivatives of the original RNN architecture include Gated Recurrency Units (GRU) [11] and LSTM (Long Short Term Memory) [17] networks. These architectures modify the base RNN approach to include some sense of longer term memory, improving the ability to process longer and more complex sequences.

5.3 Self-Attention Based Networks

Self-attention based networks have found enormous success in several AI domains, particularly within natural language processing (NLP). The Transformer archicture [30] revolutionized the field of NLP, and has spawned derivative approaches within other domains such as image analysis with the Vision Transformer [14].

Self-attention is an approach to machine learning in which a model learns to attend to different parts of the input data. Combining this self-attention mechanism with positional encoding, allows Transformers to learn how parts of an input relate to other parts of either the same input or some other input. Transformers possess the benefit of not being an iterative process, and as such can process entire sequences in a single pass greatly increasing the sample throughput compared to RNN based architectures.

5.4 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) process multidimensional inputs using multidimensional weight matricies that move across the input processing each section while reusing the same weights [2]. This is somewhat similar to how RNNs reuse their weighs for each element in a sequence, however there is no inherent ordering to which sections of the input is processed first, as such the input sections can be processed in parallel.

Due to their efficient reuse of weights, parallelizability, and multidimensional input processing ability, CNNs have found much success in the domain of image analysis.

5.5 Graph Neural Networks

Graph Neural Networks (GNN) are neural networks that operate on graph structured data. The network may operate on the graph structure itself, attributes of nodes or edges, or both.

The methods through which GNNs operate on graphs differ. In a survey by Battaglia et al. [3] several of these methods are generalized into a common framework.

The survey describe a *Graph Network Block* as the central building block of such networks. A GN Block takes a graph as input, performs computations on it using a set of update and aggregation functions, and returns a graph as output.

The update functions update node, edge and global attributes respectively; the aggregation functions aggregate data from the edges, nodes and global attributes for use in their respective update functions.

$$\begin{aligned} \phi^e &: \text{Edge update} \\ \phi^v &: \text{Node update} \\ \phi^u &: \text{Global state update} \end{aligned}$$

$$\begin{aligned} \rho^{e\to v} &: \text{Edge to node aggregation function} \\ \rho^{e\to u} &: \text{Edge to global state aggregation} \\ \rho^{v\to u} &: \text{Node to global state aggregation} \end{aligned}$$

In essence the update functions update nodes, edges and global attributes independently of each other, while the aggregation functions allow information from edges, nodes and global state to flow between eachother.

The computational steps in a so-called Full GN Block goes as follows:

1. The edges are updated.
2. Edge updates are aggregated per node.
3. The nodes are updated.
4. Edge updates are aggregated for the global state.
5. Node updates are aggregated for the global state.
6. The global state is updated.

5.5.1 Message Passing Blocks

Various types of GN blocks can be constructed from the update and aggregation functions found in Full GN blocks. One example of such blocks are Message Passing Blocks, which are the basic building blocks in the PyTorch Geometric python package that will be used in this thesis.

Message passing blocks in PyTorch Geometric consist of the following: an update function that takes a given node, as well as aggregated messages as input, and ouput a new node; an aggregation function that takes the output of the message function as input and aggregates the result; and a message function that takes node and edge information as input, and passes messages between nodes.

$$\begin{array}{rcl} X & : & \text{the set of all nodes in a graph} \\ E & : & \text{the set of all edges in a graph} \\ \gamma & : & \text{update function} \\ \square & : & \text{aggregation function} \\ \phi & : & \text{message passing function} \\ x_i^{(k)} & : & \text{node } i \text{ in layer } k \\ e_{i,j}^{(k)} & : & \text{edge from } i \text{ to } j \text{ in layer } k \\ \mathcal{N}(i) & : & \text{the neighbors of } i, \text{ i.e.: } \{j \mid \langle i, j \rangle \in E\} \end{array}$$

The message passing block can then be formulated as such:

$$x_i^{(k)} = \gamma^k \left(x_i^{(k-1)}, \square_{j \in N(i)} \phi^{(k)} \left(x_i^{(k-1)}, x_j^{(k-1)}, e_{j,i} \right) \right)$$

5.5.2 Graphsage

Graphsage is an approach to graph embedding that attempts to learn a function that generates node embeddings by sampling and aggregating features of a nodes neighborhood [16].

The node-wise formulation of the Graphsage convolution operation on node x in layer k is the following:

$$x_i^{(k)} = \Theta_1 x_i^{(k-1)} + \Theta_2 \cdot \text{mean}_{j \in \mathcal{N}(i)} x_j$$

Where Θ_1 and Θ_2 are two distinct weight matrices.

5.5.3 Graph Convoultional Networks

Graph Convolutions extend the idea regular convolutional layers, usually applied to images or time-sequence data, to graphs. Regular convolutional layers work on an area of pixels or time-slices, when applied to images or time-series' respectively. As an image may be viewed as a graph where the nodes represent pixels and the edges represent pixel adjacency, we can generalize the convolution operation to graphs. [18]

The node-wise formulation of the Graph Convolution operation on node x in layer k is the following:

$$x_i^{(k)} = \Theta^\top \sum_{j \in N(i) \cup \{i\}} \frac{ew_{i,j}}{\sqrt{\deg(i)} \cdot \sqrt{\deg(j)}} x_j^{(k-1)}$$

Where Θ, as is common in the context of deep learning functions, denotes the learnable weight matrix of the GCN layer, and with $ew_{i,j}$ denotes the edge weight from node i to node j (default: 1).

5.5.4 Relational Graph Convoultional Networks

Relational Graph Convolutional networks [29] extend the idea of GCNs (see Section 5.5.3) with the ability to process edge features in addition to node features.

The node-wise formulation of the Relational Graph Convolution operation on node x in layer k is the following:

$$x_i^{(k)} = \Theta_{\text{root}} \cdot x_i^{(k-1)} + \sum_{r \in \mathcal{R}} \sum_{j \in \mathcal{N}_r(i)} \frac{1}{|\mathcal{N}_r(i)|} \Theta_r \cdot x_j$$

Where $\mathcal{R}$ is the set of edge types and $\mathcal{N}_r(x)$ is the neighbors of x connected by edges of type r.

5.6 Hyperparameter Tuning

As the quality of many machine learning systems are highly dependent on which "hyperparameters" the systems are instantiated with. In the context of a basic feed forward deep learning model an example of its hyperparameters is the depth and width of the network. Hyperparameters may also cover meta-system parameters such as optimizer learning rate.

Hyperparameter tuning is the process of searching for a combination of hyperparameter values that produces the best performing model. A naive, and often used, approach to finding good hyperparameters is manual trial and error informed by intuition and experience. With more and more complex systems being developed however, methods that systematically and automatically search for well performing combinations of hyperparameters becomes valuable.

These more sophisticated approaches include intelligent trial scheduling where multiple models with different hyperparameters are trained in parallel, with the poorly performing models being discarded before training is finished [22]. They may also include intelligent search algorithms that attempt to find good sets of hyperparameters based on the performance of previously conducted training runs [4].

Chapter 6

Python Libraries

6.1 RDFLIB

In this thesis we use the python library rdflib to load and serialize RDF graphs stored as Turtle and XML files. Furthermore it provides a simple interface for processing graph triples directly.

6.2 PyTorch, PyTorch Lightning, and PyTorch Geometric

Throughout this project the programming language Python is used, although most of the computationally heavy operations are handled by C++ and C based libraries with Python interfaces. The libraries in question are the PyTorch deep-learning library originally created by Meta AI [28], and now part of the Linux Foundation, as well as the superset library called PyTorch Geometric. This library contains graph neural network focused expansions of the existing PyTorch library mainly aimed at rapid iteration and ease of experimentation.

6.3 RayTune

A last important library is the RayTune library. It's a suite designed with the purpose of simplifying efficient hyperparameter optimization, with implementations of common approaches such as Asynchronous Successive Halving [22] for trial scheduling and Tree-structured Parzen Estimation [4] for hyperparameter searches.

Chapter 7

Related Work

7.1 Previous Work With GNNs in General

7.1.1 'Semi-supervised classification with graph convolutional networks'

This paper tries to solve the problem of node classification employing Graph Convolution operations to embed nodes in graphs directly. The graphs considered in the paper are static and the nodes are attributed. The paper evaluates the system performance on the Citeseer, Cora, and Pubmed datasets [18].

7.1.2 'Inductive Representation Learning on Large Graphs'

This paper proposes an inductive rather than transductive approach to graph embeding, allowing it to generalize from seen to unseen nodes. Previous papers trained networks that learn specific embeddings for specific nodes, requiring all nodes to be seen during training. This paper however proposes a system (Graphsage) that learns a general function for embedding nodes [16]. The paper focuses on static graphs with attributed nodes, and evaluates it's performance on the Reddit dataset.

7.1.3 'Towards deeper graph neural networks'

This paper notes that Graph Convolutional Networks tend to fall off in performance with depth. It studies the reasoning why, and proposes a neural network based on the result of the studies called Deep Adaptive Graph Neural Network (DAGANN) [23]. They experiment on the same datasets as the original GCN paper [18].

7.1.4 'Graph attention networks'

Attention based networks have seen wide success in the realm of machine translation. This paper attempts to translate the attention models from machine translation to graph neural networks [31]. They evaluate the

success of their Graph Attention Network (GAT) on the Cora [25], Citeseer, and Pubmed datasets.

7.2 Previous Work in Graph Anomaly Detection Using GNNs

7.2.1 'One-Class Graph Neural Networks for Anomaly Detection in Attributed Networks'

This paper identifies anomalies in graphs by mapping the nodes in the graph using a graph neural network to a non-graph space in which traditional anomaly detection methods (such as Support Vector Machines) are employed [33]. The datasets employed in the paper are the recurring Cora, Citeseer, and Pubmed datasets.

7.2.2 'AddGraph: Anomaly Detection in Dynamic Graph Using Attention-based Temporal GCN.'

This paper tackles anomaly detection in dynamic graphs using a novel approach called AddGraph [34]. The basic approach outlined is to encode snapshots of a dynamic graph using a GCN, then further process the sequence of encodings with a GRU and an attention network. AddGraph is evaluated on the UCI Message dataset [26], and the Digg dataset [12].

7.2.3 'Structural Temporal Graph Neural Networks for Anomaly Detection in Dynamic Graphs'

This paper also considers anomaly detection in dynamic graphs. Their approach focuses on detecting structural changes between snapshots of the dynamic graph, as opposed to simply learning good embeddings of nodes [6]. Their approach is similar to the one outlined in the AddGraph paper [34], however they encode a subgraph of the larger graph snapshot instead of the entire snapshot, before passing the encoding to the GRU network. They evaluate their approach on the UCI Message [26] dataset, Digg [12] dataset, two bitcoin datasets [20], and several others.

Part II

The Project

Chapter 8

Problem Description

Preliminary Notation

Before defining what we mean by an anomaly, we introduce the following notation:

$\|G\|$	:	The cardinality of set G, or the number of nodes in the graph G
$a^{\underline{b}}$	:	is the falling factorial function, i.e

$$\prod_{n=0}^{b-1}(a-n)$$

8.1 Defining Structural Anomalies

First we define the precise problem we are trying to solve. As our graphs in this project we consider RDF graphs without literals and with a separation between predicates and subject/objects: for pair-wise disjoint finite sets **I**, **C**, and **P** of *IRIs* (i.e. constants), *classes* (i.e., unary predicates), and *properties* (i.e. binary predicates), a *RDF graph* is a set of *facts* of the form $C(a)$ and $P(a,b)$ where $C \in \mathbf{C}$, $P \in \mathbf{P}$, and $a, b \in \mathbf{I}$. This differs from regular RDF graphs that place no restrictions on which IRIs represent subjects, objects or predicates. Throughout this work we will also refer to these *objects* and *subjects* as *nodes*, distinct from *predicates* as *edges*.

In the definition of an anomaly below, we will use several notions. For a set of *variables* **X** disjoint from the previous sets, a *(graph) pattern* is a set of *atoms* $C(x)$ and $P(x,y)$ with $C \in \mathbf{C}$, $P \in \mathbf{P}$, and $x, y \in \mathbf{X}$. In other words, a pattern is an RDF graph where variables are used instead of IRIs. We consider patterns up to variable *renaming*—that is, see all isomorphic patterns as the same. The *distance* between patterns ρ_1 and ρ_2 is the number of atoms that are in one of them but not in the other: $\text{dist}(\rho_1, \rho_2) = |\rho_1 \setminus \rho_2| + |\rho_2 \setminus \rho_1|$.

In what follows, we essentially interpret each pattern ρ as the query Q_ρ that is the conjunction of all atoms in ρ, negations of all atoms over variables in ρ not in ρ (e.g., $\neg C(x)$ for $C(x) \notin \rho$ and $\neg P(x,y)$ for $P(x,y) \notin \rho$), and all

inequalities $x \neq y$ for x, y that are not the same in ρ. The *support* $\text{supp}(\rho, G)$ of a pattern ρ in a graph G is the number of *matches* of ρ in G—that is number of answers to Q_ρ over G.

Given a number $n \geq 0$, the *degree of n-anomaly* $\text{anom}_n(\rho, G)$ of a pattern ρ in a graph G is defined as:

$$\text{anom}_n(\rho, G) = 1 - \sum_{\rho' \colon \text{dist}(\rho, \rho') \leq n} \frac{\text{supp}(\rho', G)}{|G|^{\underline{|\rho'|}}} \tag{8.1}$$

In other words, the *n-anomaly* score of a pattern is defined as the inverse proportion of how many similar patterns to ρ, as defined by a distance threshold n, there are in the graph out of the upper bound on the possible number of patterns there can be in a given graph: $|G|^{\underline{|\rho'|}}$. The logic behind including this upper bound is to make it so that the pattern with the lowest anomaly score is the pattern that has the most possible matches in a graph of a given size.

Each set K of IRIs in a graph G has the *induced* pattern ρ_K—that is, the pattern obtained from the subgraph of G over K by replacing all IRIs by variables in a one-to-one manner. Then, the notion of the *degree of n-anomaly* $\text{anom}_n(K, G)$ of K in G for $n \geq 0$ is inherited from ρ_K. Formally: $\text{anom}_n(K, G) = \text{anom}_n(\rho_K, G)$; similarly, if ρ_K is (δ, n)-*anomalous* in G, so is K. The δ, n-*anomaly* function is defined as the following, given a pattern ρ and a graph G:

$$\text{anom}_{n,\delta}(\rho, G) = (\text{anom}_n(\rho, G) > \delta) \tag{8.2}$$

$$\text{anom}_{n,\delta}(K, G) = \text{anom}_{n,\delta}(\rho_K, G) \tag{8.3}$$

The $\text{anom}_{n,\delta}$ function returns a value of either true or false from the boolean domain $\mathbb{B}$ denoting whether a pattern is anomalous or not.

8.2 Definition of Anomaly Used in This Thesis

The definition of structural graph anomalies presented in Section 8.1 serves as a possible general interpretation of anomalous patterns in a graph.

In this thesis however, we consider a narrower subset of anomalies, namely $\text{anom}_n(\cdot, \cdot)$ functions for $n = 0$ only. The $\text{anom}_n(\cdot, \cdot)$ function can then be simplified to one that is no longer parametrised by n. That function takes a pattern ρ and a graph G as arguments and returns, as with Function 8.1, the anomaly score of that pattern in that graph:

$$\text{anom}(\rho, G) = 1 - \frac{\text{supp}(\rho, G)}{|G|^{\underline{|\rho|}}} \tag{8.4}$$

Furthermore, we consider $\text{anom}(\cdot, \cdot)$ functions that only take as inputs patterns of a size k. With k defined as the number of IRIs in the set of IRIs from which a given pattern is induced. We thus have an anomaly function parametrised by a pattern size parameter, denoted as k.

$$\text{anom}_k(\rho, G) = \text{anom}(\rho, G) \text{ defined for } \rho \text{ of size } k \tag{8.5}$$

The domain of this function is thus restricted to patterns ρ only of size k, while the range remains the set of real numbers from 0 to 1.

From this we define a new thresholded anomaly function parametrised by the threshold δ and the pattern size parameter k, taking a pattern ρ and a graph G, and returns whether or not that pattern ρ is considered anomalous in G or not:

$$\text{anom}_{\delta,k}(\rho, G) = (\text{anom}_k(\rho, G) > \delta) \tag{8.6}$$

As with Function 8.2 a set K of IRIs in a graph G is *δ-anomalous* in G if the induced pattern ρ_K is *δ-anomalous* in G.

$$\text{anom}_{\delta,k}(K, G) = \text{anom}_{\delta,k}(\rho_K, G)$$

Having formally defined what a structural anomaly in a graph is, we can turn the traditionally unsupervised machine learning problem of anomaly detection into a supervised one.

Chapter 9

Machine Learning Solution

9.1 SPARQL Based Solution

The exact anomaly detection Formula 8.6 described in Section 8.2 can be implemented using a SPARQL query based system. With the count of matches of the SPARQL query corresponding to the support function in the anomaly function. The problem with this approach however, is that the time it takes to evaluate the query grows rapidly with both pattern and graph size.

This SPARQL based system takes a graph G and a set of nodes K with $|K| = k$ as inputs and returns the $\text{anom}_{\delta,k}$ value of induced pattern of K, ρ_K. It accomplishes this by generating a query from the pattern ρ_K that ensures that the query result is the number of subgraph isomorphisms of ρ_K in G, i.e. the support of ρ_K in G.

9.1.1 Query Generation Procedure

As mentioned in Section 8.1 we consider only RDF graphs without literals and in which predicates are distinct from subjects and objects. As mentioned we will refer to subjects and objects as *nodes*, and predicates as *edges*. The *node type* of a given node is indicated by the "RDF:type" predicate in a triple, often abbreviated as "a". While *edge type* is indicated by the name of the predicate connecting two nodes.

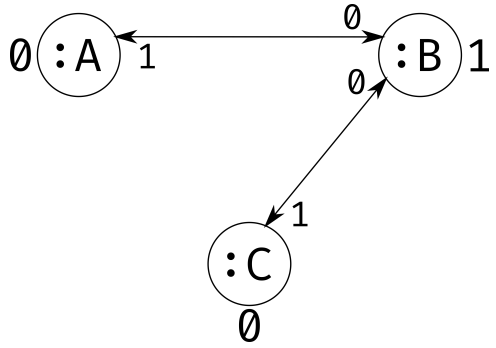

Figure 9.1: Example RDF graph pattern

The general procedure for creating a query from the induced pattern ρ_K of a set of nodes K is as follows:

1. Replace nodes in the pattern ρ_K with variables and add node type triples of those nodes.
2. Add edges between the nodes in ρ_K as subject-predicate-object triples with the edge type as the predicate.
3. Filter out matches in which nodes of the same type are not distinct nodes.
4. Filter out matches where nodes are of types in addition to the ones specified in the induced pattern ρ_K.
5. Filter out matches that contain edge connections not in ρ_K.

Thus given a pattern ρ_K consisting of three nodes taken from an RDF graph G with three different possible types of nodes, and two different possible types of edges. Depicted in figure 9.1, and as Turtle format in the following code:

```
@prefix : <http://example.org/> .

    :nodeA a :node_type0 ;
        :edge_type0 :nodeB .
    :nodeB a :node_type1 ;
        :edge_type1 :nodeA ,
                    :nodeC .
    :nodeC a :node_type0 ;
        :edge_type0 :nodeB .
```

A query for that pattern would be created as such:

1. Replace the nodes in the pattern ρ_K with variables and add the node type triples of those nodes

```
?0 a ex:node_type_0 .
?1 a ex:node_type_1 .
?2 a ex:node_type_0 .
```

2. Add the edges between the nodes in ρ_K

```
?0 ex:edge_type_0 ?1 .
?1 ex:edge_type_1 ?0 .
?1 ex:edge_type_1 ?2 .
?2 ex:edge_type_0 ?1 .
```

3. To ensure that the nodes in ρ_K are represented in the query as distinct nodes, filter out matches in which nodes of the same type are the same distinct node

```
FILTER (
    !sameTerm(?0, ?2)
)
```

4. Matches where nodes are of types in addition to the ones in ρ_K are filtered out

```
MINUS { ?0 a ex:node_type_1 . }
MINUS { ?0 a ex:node_type_2 . }
MINUS { ?1 a ex:node_type_0 . }
MINUS { ?1 a ex:node_type_2 . }
MINUS { ?2 a ex:node_type_1 . }
MINUS { ?2 a ex:node_type_2 . }
```

5. Matches that contain edge connections not in ρ_K are filtered out

```
MINUS { ?0 ex:edge_type_0 ?0 . }
MINUS { ?0 ex:edge_type_1 ?0 . }
MINUS { ?0 ex:edge_type_1 ?1 . }
MINUS { ?0 ex:edge_type_0 ?2 . }
MINUS { ?0 ex:edge_type_1 ?2 . }
MINUS { ?1 ex:edge_type_0 ?0 . }
MINUS { ?1 ex:edge_type_0 ?1 . }
MINUS { ?1 ex:edge_type_1 ?1 . }
MINUS { ?1 ex:edge_type_0 ?2 . }
MINUS { ?2 ex:edge_type_0 ?0 . }
MINUS { ?2 ex:edge_type_1 ?0 . }
MINUS { ?2 ex:edge_type_1 ?1 . }
MINUS { ?2 ex:edge_type_0 ?2 . }
MINUS { ?2 ex:edge_type_1 ?2 . }
```

6. Finally these blocks are all wrapped in the following query

```
PREFIX ex: <http://example.org/>

SELECT (count(*) as ?count) WHERE {
    # the above blocks in the order presented
}
```

9.1.2 Validation

The above SPARQL query is intended to count the number of subgraph isomorphisms of pattern ρ_K in graph G. This count is the return value of the support subfunction of our anomaly function 8.6. There are other ways to accomplish this without using SPARQL and RDF. One approach is using the python library networkx. One of networkx' sublibraries can count subgraph isomorphisms directly. When comparing the counts returned by networkx and our SPARQL approach, the counts are identical, thus verifying that our approach indeed counts the intended patterns. The networkx approach is however quite a bit slower, as such we use the SPARQL based approach in our system.

9.2 General Machine Learning Solution

The anomaly detection problem described in Section 8.1 can also be solved with a machine learning (ML) approach. In general ML systems operate in two stages, an offline training stage, and an online regression or classification stage.

During the offline stage a general ML system (in the context of supervised learning) is provided with pairs of input and outputs taken from the domain and range of some function. Through some optimization mechanism such as gradient descent it then attempts to approximate a mapping from the inputs to the outputs by iteratively altering its internal parameters, thus learning how that function maps inputs to outputs. For the online stage, the system is simply provided inputs, and returns outputs without learning and changing its internal parameters.

In order to teach an ML solution the mapping of the $\text{anom}_{\delta,k}$ function, we must provide the system with $\langle K, G \rangle$ tuples from the domain of $\text{anom}_{\delta,k}$ and the ground truth $\text{anom}_{\delta,k}(K, G)$ value of those $\langle K, G \rangle$ tuples during the offline training stage. After having learned the mapping from the domain to the range of $\text{anom}_{\delta,k}$, we can provide the system with just $\langle K, G \rangle$ tuples, and it should return approximately correct $\text{anom}_{\delta,k}$ values in the online stage.

Chapter 10

System Implementation

In this thesis we define several different systems that implement the general machine learning approach described in Section 9.2. All of the systems have in common that they consist of an *embedding* function and an *aggregation* function. The embedding function takes a graph as an input and outputs a one-to-one embeding of the feature vectors of each node in the graph. The aggregation function takes a set of node indices and a graph as inputs and aggregates the feature vectors of the nodes to produce a scalar value.

In our experiments we try 6 combinations of 3 embedding functions and 2 aggregation functions.

$\mathcal{E}_{\text{GS}}(G)$	:	A GraphSAGE-based network [16]
$\mathcal{E}_{\text{GCN}}(G)$	:	A graph convolution (GCN) network [18], also mentioned in section 5.5.3
$\mathcal{E}_{\text{RGCN}}(G)$	:	A relational GCN [29]
$\mathcal{A}_{\text{MEAN}}(K, G)$	=	$\overline{G[K]}$
$\mathcal{A}_{\text{FFN}}(K, G)$	:	a simple feed forward network that learns aggregation weights

We denote a model that combines a given embedding and aggregation function as such:

$$\mathcal{M}_{\text{GCN,FFN}}(K, G) = \mathcal{A}_{\text{FFN}}(K, \mathcal{E}_{\text{GCN}}(G))$$

10.1 Pytorch

We implement our ML system using the PyTorchLightning wrapper library for the PyTorch [28] deep learning library. As our loss function we employ binary cross entropy and as our optimizer we use AdamW [24].

10.1.1 General Prediction

As the offline real-time prediction case the system takes a graph: G, and a set of node indicies K from the graph. First it embeds G using it's embedding function:

$$\mathcal{E}(G) = \widehat{G}$$

It then aggregates the embedded values of the nodes indices K in $\hat{G}$ and returns the systems guess at the $\text{anom}_{\delta,k}$ value of the pattern induced by those K in G:

$$\mathcal{A}(K, \widehat{G}) = \widehat{\text{anom}_{\delta,k}}$$

10.1.2 Training Phase

During training the system takes batches consisting of a graph G and a list of multiple sets of nodes $\mathbf{K}$ and the $\text{anom}_{\delta,k}$ for the induced pattern of each $K \in \mathbf{K}$. As outlined in Section 10.1.1 the system embeds the graph G and aggregates the values of each set of nodes $K \in \mathbf{K}$ in the embedded graph $\widehat{G}$. Rather than returning a binary "anomalous" or "non-anomalous" value as in Section 10.1.1 however, it instead calculates the binary cross entropy (BCE) of it's output distribution and the ground truth $\text{anom}_{\delta,k}$ distribution provided as an input. This BCE score is passed to the AdamW optimizer [24] which updates the internal model weights so that BCE is minimized.

10.1.3 Validation and Testing

During validation and testing the system also receives batches the form described in Section 10.1.2, furthermore it also embeds and aggregates the batch in the same way. Instead of calculating the BCE score of the output and intput distribution however, it calculates and returns several performace metrics described in further detail in Section 11.3.

Chapter 11

Benchmarks

To compare our systems' performance with the accurate but slow (for larger graphs) SPARQL based method, as well as with each other, we need benchmarks. Our benchmarks are sets of pairs of inputs and expected outputs. Since we are attempting to teach our system the $\text{anom}_{\delta,k}$ function, our inputs are from the domain of $\text{anom}_{\delta,k}$, that is tuples of the form $\langle K, g \rangle$ with K being a set of nodes of size k from the graph g. The expected outputs are from the range of $\text{anom}_{\delta,k}$, that is: either 0 or 1, indicating whether a pattern induced by the nodes K in a graph g is anomalous or not. Thus our benchmarks are sets of tuples of the form $\langle \langle K, g \rangle , \text{anom}_{\delta,k}(K, g) \rangle$.

Because our SPAQRL based solution is feasibly fast on smaller graphs, but prohibitively slow on larger ones, we can use it to generate the $\text{anom}_{\delta,k}$ scores for our benchmarks, as long as the graph and pattern induced by the set of nodes are small enough. Ideally however, we would like our system to be able to classify sets of nodes in graphs larger than what is feasibly classifiable using our SPARQL solution. In order to measure the extent to which it is possible for our system to generalize from experience with small graphs to larger ones, we can train our systems on sets of nodes from smaller subgraphs of a large graph G, and test our systems on sets of nodes from larger subgraphs of that same large graph G.

11.1 General Benchmark Creation Procedure

G	:	A large RDF graph without literals in which predicates (edges) are separate from subjects and objects (nodes)
δ	:	A real number denoting the threshold for what is considered anomalous
k	:	A positive integer denoting the allowed size of the sets of nodes
$\mathbf{s}$	:	A multiset of integer values denoting the size of each subgraph to be sampled from G
N_K	:	An integer denoting the number of sets of nodes from which we induce patterns to be sampled from each subgraph of G.

Given these parameters our general benchmark creation procedure is the

following:

1. For each $s \in \mathbf{s}$ sample (procedure described in Section 11.1.1) a subgraph from G of size s. This set of subgraphs is denoted $\mathbf{g}$.
2. For each $g_i \in \mathbf{g}$ sample (procedure described in Section 11.1.2) a list of sets of nodes $\mathbf{K}_i$ of length N_K, with each $K_{ij} \in \mathbf{K}_i$ being of size k.
   (a) For each $K_{ij} \in \mathbf{K}_i$, construct the $\langle\langle K_{ij}, g_i\rangle, \text{anom}_{\delta,k}(K_{ij}, g_i)\rangle$ input-output tuples using our SPARQL method.
3. Finally as we would like our system to only train on smaller subgraphs, and test on unseen larger subgraphs, we split our set of input-output tuples according to the size of the subgraphs they are derived from.

   The input-output tuples derived from the largest subgraphs are reserved for testing and validation, while the smaller ones are reserved for training.

11.1.1 Subgraph Sampling Procedure

The procedure for sampling the subgraphs $\mathbf{g}$ from G is essentially a breadth-first search.

Given the following parameters:

G : A graph from which we sample subgraphs
s : An integer denoting the size of the subgraph to be sampled

The subgraph sampling procedure is as such:

1. Begin with a randomly sampled node from G and add it to the set of visited nodes V.
2. For each node $x \in V$ add it's neighborhood $\mathcal{N}(x)$, defined as the nodes it's connected to in G, to the set of visited nodes V.
3. Repeat step 2 until there are no nodes left to visit in G, or until the number of visited nodes $|V|$ is equal to the desired number of sampled nodes s. If the number of nodes added in the last step causes the total number of nodes to exceed the desired size s, discard a random set of nodes added in the last iteration equal to the surplus amount.
4. Return the subgraph induced by V.

11.1.2 Node Set Sampling Procedure

The sampling procedure for the sets of nodes is similar to the one described in section 11.1.1, however it differs in that nodes are not sampled through a breadth-first search, instead they are sampled as they are visited in a random walk. Given the following parameters:

g : A graph from which we sample sets of nodes.
k : An integer denoting the size of the sets of nodes to be sampled from a graph g.

The pattern sampling procedure is as such:

1. Begin with a randomly sampled node from g and add it to the list of visited nodes V.
2. Sample a random node from the set of the nodes neighboring the visited nodes $\{\mathcal{N}(x) \mid x \in V\}$ and add it to the set of visited nodes V.
3. Repeat until there are no nodes left to visit in g or until the number of visited nodes $|V|$ is equal to the desired number of nodes k.
4. Return the visited nodes V.

11.2 Concrete Benchmarks

In this thesis we base our benchmarks on three publically available graphs: the IMDB graph [15] and the LastFM [15] graph, as well as a graph generated using the Lehigh University Benchmark graph generator, referred to as LUBM [21] (see Chapter 3). These three graphs are our large graphs G in the context of our benchmark creation procedure described in Section 11.1. For each of these three graphs, benchmarks use sets of nodes of size k as 2 and 3. For each of the combinations of k and G we consider two values of δ. These values for δ were chosen such that they split the training, validation and test sets of each benchmark in such a way that there are always both anomalous and non-anomalous samples in all subsets. These values for δ were found using trial and error. In total that leaves us with the 12 benchmarks listed in Table 11.2.

A given benchmark will from here on be uniquely identified in text in the following manner: k2-IMDB-δ1.1 referring to the first benchmark listed in Table 11.2.

11.2.1 Subgraph sizes

Due to the fact that we create the ground truth values for our samples using the SPARQL based method described in Section 9.1, the subgraphs sizes we sample (i.e **s**), are choosen so that the SPARQL based method runs in a resonable amount of time.

The number of sets of nodes N_K was chosen to be large enough so that the likleyhood of representing most of the variety of possible patterns in the graph from which sample is high. Also due to the fact that the PyTorch implementation of our system can process the list of sets of nodes **K** in parallel, the upper limit of the size of **K** is how many sets of nodes can be held in memory at the same time. That is why we limit N_K to 128 in this thesis.

The sizes of the subgraphs sampled from the LastFM graph is smaller than the ones sampled from the IMDB graph. The reason for this is based on trial and error testing in which the SPARQL procedure runs markedly slower on LastFM subgraphs compared to IMDB ones. This is likely due to the fact that the LastFM has a much higher edge-to-node ratio than IMDB

(the number of edges and nodes in each respective graph is shown in Table 11.1).

Also, the sizes **s** listed for LUBM in Table 11.2 is expressed in terms of the number of departments included in the generated graph (see Section 3.1 for a brief description of the LUBM graph structure).

Dataset	# Nodes	# Edges	# Node Types	# Edge Types
IMDB	11616	34212	3	4
LastFM	20612	201908	3	5
LUBM	Variable	Variable	14	12

Table 11.1: The number of nodes and edges, as well as node and edge types in our graphs

k	Dataset	δ	**s**, with 32 g of each size	N_K
2	IMDB	1.1	$[512, 1024, 2048, 4096, 4096]$	128
		1.2		
2	LUBM	0.6	$[1, 2, 4, 8, 8]$	128
		0.7		
2	LastFM	0.4	$[256, 512, 768, 1024, 1024]$	128
		0.7		
3	IMDB	0.6	$[512, 1024, 2048, 4096, 4096]$	128
		0.625		
3	LUBM	0.425	$[1, 2, 4, 8, 8]$	128
		0.5		
3	LastFM	0.6	$[256, 512, 768, 1024, 1024]$	128
		0.8		

Table 11.2: Our 12 benchmarks. Values for δ is expressed in terms of standard deviations of anom_k (defined in equation 8.5) scores standardized by the mean and standard deviation of the training set.

11.3 Metrics

When comparing our systems we care about two categories of metrics: The quality of the anomaly detection, and the time it takes to perform.

11.3.1 Anomaly Detection Quality

For classification problems there are many metrics that can be used. Among the most common ones we have: accuracy and F1 score. The simple accuracy measure is often misleading in cases where there are gross class imbalances. F1 solves this, however it is often too optimistic in cases where negative classification performance is poor. For this reason we will primarily use the Matthews Correlation Coefficient (MCC) metric instead. It solves the

problems of both accuracy and F1 measures, reflecting classification issues that both accuracy and F1 measures miss [9].

As our systems internally produce real valued outputs ranging from 0 to 1; only thresholding them at 0.5 before returning a binary prediction. We can also evaluate classification performance across a range of classification thresholds, instead of just 0.5 which is the case with MCC. To do this we use the secondary classification metric Average Precision (AP), defined as the area under the precision-recall curve created by measuring precision and recall at multiple thresholds.

Definitions

MCC is calculated as an aggregate of all the four 1 scores in a binary confusion matrix in the following manner:

$$MCC = \frac{TP \cdot TN - FP \cdot FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

MCC score ranges from -1 to +1; +1 represents perfect classification of both negative and positive samples, while -1 represents no correctly classified samples at all. Finally, a score of 0 means the model performs no better than a random classifier.

AP is calculated as the area under the precision-recall curve, this curve plots precision against recall at different classification thresholds. AP ranges from 0 to 1, with 1 representing a perfect classifier, and 0 representing a completely inept one. The AP performance of a random classifier equals the proportion of positive samples in the benchmark.

11.3.2 Inference Time

We report the inference time of our system in terms of sample throughput. That is, the number of sets of nodes classified per unit of time. SPARQL throughput is measured on the CPU with access to 6 parallel threads as that is how many threads are available in our environment. We restrict our machine learning approach to the same number of threads. This provides us with a one-to-one comparison of the SPAQRL based and ML based approaches given the same operating environment. In a realistic machine learning setting however, one would likely have access to GPUs. As such we also measure the throughput using GPUs. This is not a one to one comparison in terms of algorithmic performance, however it serves as a good comparison in terms of realistically achievable speed up in a real world setting.

Part III

Conclusion

Chapter 12

Evaluation

For each of the benchmarks listed in Table 11.2 we experiment with several systems as mentioned in Chapter 10. We have 6 combinations of embedding and aggregation functions, with 12 benchmarks the total number of experiments thus becomes 72.

12.1 Hyperparameter Tuning

In order to measure the quality of our systems on the benchmarks, we train them using the hyperparameter optimization library RayTune. Specifically we use HyperOptSearch (based on Tree-Structured Parzen Estimation [4]) as our searching algorithm, and the ASHA [22] as our trial scheduler. We train for a maximum of 8 epochs, and sample 16 combinations of the the hyperparameters listed in Table 12.1 for each of our benchmarks.

12.2 Results

The results after running the 72 experiments described in Section 12 are listed in Tables 12.2 and 12.3, for the benchmarks with $k = 2$ and $k = 3$. When referencing concrete benchmarks later in this text we will do so in the following manner: the benchmark created from the IMDB dataset with $k = 2$ and a δ threshold of 0.6 will be referred to as "k2-IMDB-δ0.6".

Variable	Range
Learning rate	$[5e-6, 5e-3]$
Positive sample loss weighting	$[0.1, 10]$
Embedding network width	$[8, 1024]$
Embedding network depth	$[2, 10]$
Aggregation network width (Null for $\mathcal{A}_{\text{mean}}$)	$[8, 1024]$
Aggregation network depth (Null for $\mathcal{A}_{\text{mean}}$)	$[2, 10]$

Table 12.1: Variables that RayTune searches over, and the ranges from which the variable values are sampled.

k	Dataset	δ	Model	MCC	F1	AP	PR	RC	TP	TN
2	IMDB	1.1	$\mathcal{M}_{\text{GCN, FFN}}$	0.00	0.00	0.65	0.00	0.00	0.00	1.00
			$\mathcal{M}_{\text{GCN, MEAN}}$	0.64	0.72	0.94	1.00	0.56	0.56	1.00
			$\mathcal{M}_{\text{GS, FFN}}$	0.74	0.82	0.94	1.00	0.69	0.69	1.00
			$\mathcal{M}_{\text{GS, MEAN}}$	0.74	0.82	0.88	1.00	0.69	0.69	1.00
			$\mathcal{M}_{\text{RGCN, FFN}}$	0.74	0.82	0.83	1.00	0.69	0.69	1.00
			$\mathcal{M}_{\text{RGCN, MEAN}}$	0.74	0.82	0.86	1.00	0.69	0.69	1.00
		1.2	$\mathcal{M}_{\text{GCN, FFN}}$	0.00	0.00	0.31	0.00	0.00	0.00	1.00
			$\mathcal{M}_{\text{GCN, MEAN}}$	0.00	0.00	0.59	0.00	0.00	0.00	1.00
			$\mathcal{M}_{\text{GS, FFN}}$	1.00	1.00	1.00	1.00	1.00	1.00	1.00
			$\mathcal{M}_{\text{GS, MEAN}}$	1.00	1.00	1.00	1.00	1.00	1.00	1.00
			$\mathcal{M}_{\text{RGCN, FFN}}$	1.00	1.00	1.00	1.00	1.00	1.00	1.00
			$\mathcal{M}_{\text{RGCN, MEAN}}$	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	LastFm	**0.4**	$\mathcal{M}_{\text{GCN, FFN}}$	**0.51**	**0.80**	**0.91**	**0.69**	**0.96**	**0.96**	**0.48**
			$\mathcal{M}_{\text{GCN, MEAN}}$	**0.33**	**0.48**	**0.72**	**0.86**	**0.33**	**0.33**	**0.94**
			$\mathcal{M}_{\text{GS, FFN}}$	**0.66**	**0.85**	**0.94**	**0.81**	**0.90**	**0.90**	**0.75**
			$\mathcal{M}_{\text{GS, MEAN}}$	**0.66**	**0.85**	**0.95**	**0.80**	**0.91**	**0.91**	**0.73**
			$\mathcal{M}_{\text{RGCN, FFN}}$	**0.64**	**0.84**	**0.90**	**0.81**	**0.87**	**0.87**	**0.77**
			$\mathcal{M}_{\text{RGCN, MEAN}}$	**0.64**	**0.84**	**0.95**	**0.81**	**0.87**	**0.87**	**0.77**
		0.7	$\mathcal{M}_{\text{GCN, FFN}}$	**0.53**	**0.72**	**0.81**	**0.58**	**0.95**	**0.95**	**0.59**
			$\mathcal{M}_{\text{GCN, MEAN}}$	**0.29**	**0.46**	**0.60**	**0.66**	**0.35**	**0.35**	**0.89**
			$\mathcal{M}_{\text{GS, FFN}}$	**0.56**	**0.74**	**0.60**	**0.61**	**0.94**	**0.94**	**0.63**
			$\mathcal{M}_{\text{GS, MEAN}}$	**0.57**	**0.74**	**0.78**	**0.62**	**0.94**	**0.94**	**0.64**
			$\mathcal{M}_{\text{RGCN, FFN}}$	**0.57**	**0.74**	**0.81**	**0.62**	**0.94**	**0.94**	**0.64**
			$\mathcal{M}_{\text{RGCN, MEAN}}$	**0.57**	**0.74**	**0.76**	**0.62**	**0.94**	**0.94**	**0.64**
	LUBM	0.6	$\mathcal{M}_{\text{GCN, FFN}}$	1.00	1.00	1.00	1.00	1.00	1.00	1.00
			$\mathcal{M}_{\text{GCN, MEAN}}$	0.93	0.97	1.00	0.98	0.97	0.97	0.96
			$\mathcal{M}_{\text{GS, FFN}}$	1.00	1.00	1.00	1.00	1.00	1.00	1.00
			$\mathcal{M}_{\text{GS, MEAN}}$	1.00	1.00	1.00	1.00	1.00	1.00	1.00
			$\mathcal{M}_{\text{RGCN, FFN}}$	1.00	1.00	1.00	1.00	1.00	1.00	1.00
			$\mathcal{M}_{\text{RGCN, MEAN}}$	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		0.7	$\mathcal{M}_{\text{GCN, FFN}}$	0.65	0.73	0.89	0.85	0.65	0.65	0.95
			$\mathcal{M}_{\text{GCN, MEAN}}$	0.35	0.41	0.66	0.76	0.28	0.28	0.96
			$\mathcal{M}_{\text{GS, FFN}}$	0.86	0.89	0.89	0.98	0.81	0.81	0.99
			$\mathcal{M}_{\text{GS, MEAN}}$	0.69	0.72	0.88	1.00	0.56	0.56	1.00
			$\mathcal{M}_{\text{RGCN, FFN}}$	0.69	0.72	0.89	1.00	0.56	0.56	1.00
			$\mathcal{M}_{\text{RGCN, MEAN}}$	0.68	0.71	0.84	1.00	0.55	0.55	1.00

Table 12.2: Results for all benchmarks with $k = 2$. PR, RC, TP, and TN corresponds to precision, recall, true positive rate, and true negative rate respectively.

k	Dataset	δ	Model	MCC	F1	AP	PR	RC	TP	TN
3	IMDB	0.600	$\mathcal{M}_{\text{GCN, FFN}}$	0.62	0.88	0.95	0.86	0.89	0.89	0.72
			$\mathcal{M}_{\text{GCN, MEAN}}$	0.49	0.83	0.89	0.82	0.83	0.83	0.65
			$\mathcal{M}_{\text{GS, FFN}}$	0.87	0.95	0.97	1.00	0.90	0.90	1.00
			$\mathcal{M}_{\text{GS, MEAN}}$	0.84	0.94	0.99	0.99	0.89	0.89	0.98
			$\mathcal{M}_{\text{RGCN, FFN}}$	0.87	0.95	0.99	1.00	0.90	0.90	1.00
			$\mathcal{M}_{\text{RGCN, MEAN}}$	0.87	0.95	1.00	1.00	0.90	0.90	1.00
		0.625	$\mathcal{M}_{\text{GCN, FFN}}$	0.71	0.85	0.95	0.96	0.76	0.76	0.95
			$\mathcal{M}_{\text{GCN, MEAN}}$	0.24	0.30	0.79	0.92	0.18	0.18	0.98
			$\mathcal{M}_{\text{GS, FFN}}$	0.76	0.87	0.97	1.00	0.77	0.77	1.00
			$\mathcal{M}_{\text{GS, MEAN}}$	0.76	0.87	0.95	1.00	0.77	0.77	1.00
			$\mathcal{M}_{\text{RGCN, FFN}}$	0.82	0.91	0.99	1.00	0.83	0.83	1.00
			$\mathcal{M}_{\text{RGCN, MEAN}}$	0.54	0.66	1.00	1.00	0.50	0.50	1.00
	LastFm	**0.600**	$\mathcal{M}_{\text{GCN, FFN}}$	**0.38**	**0.69**	**0.78**	**0.66**	**0.74**	**0.74**	**0.65**
			$\mathcal{M}_{\text{GCN, MEAN}}$	**0.32**	**0.70**	**0.68**	**0.53**	**1.00**	**1.00**	**0.20**
			$\mathcal{M}_{\text{GS, FFN}}$	**0.47**	**0.74**	**0.80**	**0.69**	**0.78**	**0.78**	**0.68**
			$\mathcal{M}_{\text{GS, MEAN}}$	**0.34**	**0.70**	**0.60**	**0.54**	**1.00**	**1.00**	**0.22**
			$\mathcal{M}_{\text{RGCN, FFN}}$	**0.50**	**0.76**	**0.80**	**0.68**	**0.87**	**0.87**	**0.62**
			$\mathcal{M}_{\text{RGCN, MEAN}}$	**0.37**	**0.69**	**0.81**	**0.65**	**0.74**	**0.74**	**0.63**
		0.800	$\mathcal{M}_{\text{GCN, FFN}}$	0.35	0.38	0.52	0.69	0.27	0.27	0.97
			$\mathcal{M}_{\text{GCN, MEAN}}$	0.09	0.06	0.27	0.50	0.03	0.03	0.99
			$\mathcal{M}_{\text{GS, FFN}}$	0.56	0.66	0.60	0.57	0.78	0.78	0.85
			$\mathcal{M}_{\text{GS, MEAN}}$	0.40	0.54	0.46	0.47	0.62	0.62	0.83
			$\mathcal{M}_{\text{RGCN, FFN}}$	0.56	0.65	0.52	0.64	0.67	0.67	0.90
			$\mathcal{M}_{\text{RGCN, MEAN}}$	0.52	0.62	0.72	0.50	0.81	0.81	0.80
	LUBM	0.425	$\mathcal{M}_{\text{GCN, FFN}}$	0.60	0.87	0.97	1.00	0.77	0.77	1.00
			$\mathcal{M}_{\text{GCN, MEAN}}$	0.93	0.99	1.00	1.00	0.98	0.98	1.00
			$\mathcal{M}_{\text{GS, FFN}}$	0.60	0.87	1.00	1.00	0.77	0.77	1.00
			$\mathcal{M}_{\text{GS, MEAN}}$	0.60	0.87	1.00	1.00	0.77	0.77	1.00
			$\mathcal{M}_{\text{RGCN, FFN}}$	0.60	0.87	0.99	1.00	0.77	0.77	1.00
			$\mathcal{M}_{\text{RGCN, MEAN}}$	0.60	0.87	1.00	1.00	0.77	0.77	1.00
		0.500	$\mathcal{M}_{\text{GCN, FFN}}$	0.45	0.43	0.80	0.93	0.28	0.28	0.99
			$\mathcal{M}_{\text{GCN, MEAN}}$	0.22	0.12	0.42	0.98	0.07	0.07	1.00
			$\mathcal{M}_{\text{GS, FFN}}$	0.56	0.63	0.82	0.79	0.53	0.53	0.95
			$\mathcal{M}_{\text{GS, MEAN}}$	0.54	0.54	0.87	0.95	0.38	0.38	0.99
			$\mathcal{M}_{\text{RGCN, FFN}}$	0.54	0.53	0.86	0.97	0.37	0.37	1.00
			$\mathcal{M}_{\text{RGCN, MEAN}}$	0.54	0.53	0.85	1.00	0.36	0.36	1.00

Table 12.3: Results for all benchmarks with $k = 3$. PR, RC, TP, and TN corresponds to precision, recall, true positive rate, and true negative rate respectively.

k	Dataset	δ	Model	SPARQL	ML CPU	ML GPU
2	IMDB	1.1	$\mathcal{M}_{\text{GCN, FFN}}$	40	805	985
			$\mathcal{M}_{\text{GCN, MEAN}}$	40	1797	11107
			$\mathcal{M}_{\text{GS, FFN}}$	40	26384	10257
			$\mathcal{M}_{\text{GS, MEAN}}$	40	5744	25921
			$\mathcal{M}_{\text{RGCN, FFN}}$	40	9695	6335
			$\mathcal{M}_{\text{RGCN, MEAN}}$	40	7836	3497
		1.2	$\mathcal{M}_{\text{GCN, FFN}}$	40	389	487
			$\mathcal{M}_{\text{GCN, MEAN}}$	40	9865	4971
			$\mathcal{M}_{\text{GS, FFN}}$	40	2675	2592
			$\mathcal{M}_{\text{GS, MEAN}}$	40	637	6338
			$\mathcal{M}_{\text{RGCN, FFN}}$	40	828	1545
			$\mathcal{M}_{\text{RGCN, MEAN}}$	40	4872	9358
	LastFm	0.4	$\mathcal{M}_{\text{GCN, FFN}}$	36	5421	3040
			$\mathcal{M}_{\text{GCN, MEAN}}$	36	18540	11329
			$\mathcal{M}_{\text{GS, FFN}}$	36	2009	1095
			$\mathcal{M}_{\text{GS, MEAN}}$	36	7623	15470
			$\mathcal{M}_{\text{RGCN, FFN}}$	36	6192	1967
			$\mathcal{M}_{\text{RGCN, MEAN}}$	36	813	3395
		0.7	$\mathcal{M}_{\text{GCN, FFN}}$	36	10478	3148
			$\mathcal{M}_{\text{GCN, MEAN}}$	36	12429	10342
			$\mathcal{M}_{\text{GS, FFN}}$	36	2444	589
			$\mathcal{M}_{\text{GS, MEAN}}$	36	10370	9723
			$\mathcal{M}_{\text{RGCN, FFN}}$	36	10610	2940
			$\mathcal{M}_{\text{RGCN, MEAN}}$	36	673	2076
	LUBM	0.6	$\mathcal{M}_{\text{GCN, FFN}}$	20	107	323
			$\mathcal{M}_{\text{GCN, MEAN}}$	20	4630	9292
			$\mathcal{M}_{\text{GS, FFN}}$	20	78	283
			$\mathcal{M}_{\text{GS, MEAN}}$	20	166	2302
			$\mathcal{M}_{\text{RGCN, FFN}}$	20	1938	1171
			$\mathcal{M}_{\text{RGCN, MEAN}}$	20	245	893
		0.7	$\mathcal{M}_{\text{GCN, FFN}}$	20	695	722
			$\mathcal{M}_{\text{GCN, MEAN}}$	20	1450	8702
			$\mathcal{M}_{\text{GS, FFN}}$	20	140	301
			$\mathcal{M}_{\text{GS, MEAN}}$	20	533	5728
			$\mathcal{M}_{\text{RGCN, FFN}}$	20	1661	1499
			$\mathcal{M}_{\text{RGCN, MEAN}}$	20	2381	1276

Table 12.4: Sample throughput for all benchmarks expressed in samples processed per second, for $k = 2$.

k	Dataset	δ	Model	SPARQL	ML CPU	ML GPU
3	IMDB	0.600	$\mathcal{M}_{\text{GCN, FFN}}$	5	1201	2551
			$\mathcal{M}_{\text{GCN, MEAN}}$	5	2870	6286
			$\mathcal{M}_{\text{GS, FFN}}$	5	191	486
			$\mathcal{M}_{\text{GS, MEAN}}$	5	16227	20812
			$\mathcal{M}_{\text{RGCN, FFN}}$	5	457	1001
			$\mathcal{M}_{\text{RGCN, MEAN}}$	5	150	1191
		0.625	$\mathcal{M}_{\text{GCN, FFN}}$	5	23979	10422
			$\mathcal{M}_{\text{GCN, MEAN}}$	5	2793	5142
			$\mathcal{M}_{\text{GS, FFN}}$	5	7117	2657
			$\mathcal{M}_{\text{GS, MEAN}}$	5	3194	10300
			$\mathcal{M}_{\text{RGCN, FFN}}$	5	5006	2826
			$\mathcal{M}_{\text{RGCN, MEAN}}$	5	103	915
	LastFm	0.600	$\mathcal{M}_{\text{GCN, FFN}}$	1	10354	8241
			$\mathcal{M}_{\text{GCN, MEAN}}$	1	3535	5352
			$\mathcal{M}_{\text{GS, FFN}}$	1	454	803
			$\mathcal{M}_{\text{GS, MEAN}}$	1	13912	13546
			$\mathcal{M}_{\text{RGCN, FFN}}$	1	2625	1437
			$\mathcal{M}_{\text{RGCN, MEAN}}$	1	7471	2797
		0.800	$\mathcal{M}_{\text{GCN, FFN}}$	1	382	931
			$\mathcal{M}_{\text{GCN, MEAN}}$	1	11069	8786
			$\mathcal{M}_{\text{GS, FFN}}$	1	9701	5338
			$\mathcal{M}_{\text{GS, MEAN}}$	1	13997	15535
			$\mathcal{M}_{\text{RGCN, FFN}}$	1	3937	1659
			$\mathcal{M}_{\text{RGCN, MEAN}}$	1	3480	2129
	LUBM	0.425	$\mathcal{M}_{\text{GCN, FFN}}$	6	2384	2242
			$\mathcal{M}_{\text{GCN, MEAN}}$	6	590	4483
			$\mathcal{M}_{\text{GS, FFN}}$	6	9460	8183
			$\mathcal{M}_{\text{GS, MEAN}}$	6	7388	14325
			$\mathcal{M}_{\text{RGCN, FFN}}$	6	3181	1695
			$\mathcal{M}_{\text{RGCN, MEAN}}$	6	229	1593
		0.500	$\mathcal{M}_{\text{GCN, FFN}}$	6	7075	5368
			$\mathcal{M}_{\text{GCN, MEAN}}$	6	370	2621
			$\mathcal{M}_{\text{GS, FFN}}$	6	14205	8377
			$\mathcal{M}_{\text{GS, MEAN}}$	6	148	2060
			$\mathcal{M}_{\text{RGCN, FFN}}$	6	1383	797
			$\mathcal{M}_{\text{RGCN, MEAN}}$	6	166	1388

Table 12.5: Sample throughput for all benchmarks expressed in samples processed per second, for $k = 3$.

12.2.1 Classification Quality

As mentioned in Section 11.3 regarding metrics, we measure our ML system's classifaction performance in terms of MCC (Matthews Correlation Coefficient) and AP (average precision). In regards to classification performance we see some trends in our results.

For the k2-IMDB-δ1.2 and k2-LUBM-δ0.6 benchmarks we see perfect 1.0 MCC scores. In the context of machine learning, perfect scores are often suspicious in that they more often than not indicate bugs as opposed to a good classifier.

There are also instances where differing models perform identically despite not scoring perfectly. For example for the k2-IMDB-δ1.1 benchmark, all the Graphsage and RGCN based models have an identical MCC score of 0.743, albeit with different average precision scores. This phenomenon of the Graphsage and RGCN models performing almost identically, while GCN lags behind can be seen throughout the results for benchmarks:

1. k2-LastFM-δ0.4
2. k2-LastFM-δ0.7
3. k3-IMDB-δ0.6
4. k3-LUBM-δ0.425

12.2.2 Throughput

As can be observed the SPARQL throughput remains identical for a given combination of k and dataset, irrespective of our thresholding value δ. This is because in our anomaly scoring function the SPARQL approach only replaces the $\text{supp}(K, G)$ subfunction, which does not depend on δ at all.

In terms of throughput comparison, our ML approach running on a CPU is significantly faster in all cases, with the SPARQL approach only coming relatively close in speed in the case of $k = 2$ LUBM with $\delta = 0.6$, with $\mathcal{M}_{\text{gs, ffn}}$. Still, in that particular case the ML approach is almost 4 times faster. In most other cases it is 10s of times faster.

An interesting, but ultimately expected, observation is the throughput performance for our ML system running on the GPU: it is not always the case that it outperforms the CPU performance. A likely explanation is that some ML models, or the same model with different hyperparameters increasing the computational burden, simply benefit more from GPU acceleration than others.

12.3 Discussion

In order to explain the cases perfect MCC score performance it may be the case that the samples are easily identifiable as negative or positive by a simple feature of the data. For example: all positive patterns include nodes of type 2, while all negative patterns include nodes of type 1.

An alternative explanation is that the model in question is complex enough, or that the number of unique patterns is low enough, for the model to have memorized every single pattern. In the context of machine learning this normally leads to overfitting in which the model generalizes poorly from seen to unseen data. This is normally controlled for by using disjunct training and validation sets. In our case however, due to the way in which we sample our patterns (see Section 11.1.2), there is the possibility of a given pattern begin found both in the training and validation sets. In fact, there is also a posibility of a given pattern to be found in both the negative and positive splits of the training or test sets. We will look into this further in Section 12.3.2.

12.3.1 Confusion Matrix

In order to further investigate the source of these perfect scores I first verified that the MCC calculation was correct by looking at confusion matrix scores also listed in Tables 12.2 12.3. This also gives us an opportunity to narrow down whether performance is skewed towards positive or negative sample classification. As a note: Only true positive (TP) and true negative (TN) rates are listed as this is binary classification problem and false positive (FP) and false negative (FN) rates are simply the complements of TP and TN respectively.

Looking at the TP and TN scores directly rather than just the aggregate MCC score we can discern the following additional trend: TN performance is often perfect even when TP is not, this signifies that negative samples are more easily classifiable than positive ones. An exception is in the cases of $k2$-LastFM and $k3$-LastFM-$\delta 0.6$, in those cases the system classifies positive samples with a higher accuracy than negative ones. It happens occasionally otherwise as well, but not across an entire benchmark as with LastFM.

12.3.2 Pattern Overlap

In Section 12.3 we presented two possible explanations for the high performance: The patterns in our examples may all possess some feature that makes them trivially classifiable, or the patterns might all be "memorized" by our model, either because the unique patterns are few in number, or because our model is sufficiently complex. In order to investigate the latter hypothesis we look at the degree of which we find the patterns in our training sets, in our test sets.

The procedure for checking the overlap of patterns in the test training set is as follows:

1. We first split our test set in its negative and positive examples.
2. We then filter out all isomorphic patterns from the test and training sets, both to save time when looking for matches, and to count the number of unique patterns.
3. We then count the number of patterns from the test set that are also found in the training set using isomorphism checks. These counts are

k	Dataset	δ	Negative	Positive
2	IMDB	1.1	1.000	1.000
		1.2	1.000	1.000
	LUBM	0.6	1.000	1.000
		0.7	1.000	1.000
	LastFM	0.4	1.000	1.000
		0.7	1.000	1.000
3	IMDB	0.6	1.000	1.000
		0.625	1.000	1.000
	LUBM	**0.425**	**1.000**	**0.976**
		0.5	**0.994**	**0.939**
	LastFM	0.6	1.000	1.000
		0.8	1.000	1.000

Table 12.6: Pattern overlap between negative and positive test examples in the full training sets

then weighted by the number of isomorphic patterns initially filtered out of the subset.

Looking at the results of this procedure listed in Table 12.6, the degree of overlap between the patterns in the test sets in the training sets is almost always 100%, both for the negative and positive test examples. Also the number of unique patterns, shown in Table 12.7 is low with the exception of the LUBM benchmarks.

This large degree of overlap between the patterns in the test sets and in the training sets is not that surprising given that our pattern sampling procedure (see Section 11.1.2) simply samples a random neighborhood of size k. The only difference between our training and test sets is that the patterns in the training sets are sampled from smaller subgraphs of the large graph G, while the patterns in the test sets are sampled from larger subgraphs of the same large graph G. The fewer the possible unique patterns of size k in the large graph G, the higher likelyhood that the same pattern will be sampled multiple times, and that overlap will increase.

The Table 12.7 lists the number of unique patterns in each benchmark. It shows that the number of unique patterns is indeed quite low with the exception of LUBM. And as expected, the only cases where the overlap in patterns from the test and training sets is not 100% (Table 12.6), is the LUBM cases where there are many unique patterns relative to the other benchmarks.

In this table we can also see that the total number of unique negative and positive patterns in the test set sum up to more than the total amount of unique patterns. This indicates an overlap between the patterns in the negative and postive classes within just the test sets as well. The fact that the same pattern can both be positive and negative, seems contradictory, however, this is possible as our measure of an anomaly not only depends

k	Dataset	δ	Negative	Positive	All
2	IMDB	1.1	1	2	2
		1.2	1	1	2
	LUBM	**0.6**	**2**	**57**	**59**
		0.7	**6**	**53**	**59**
	LastFm	0.4	2	3	3
		0.7	2	3	3
3	IMDB	0.6	1	4	4
		0.625	1	3	4
	LUBM	**0.425**	**1**	**370**	**370**
		0.5	**100**	**289**	**370**
	LastFm	0.6	5	9	9
		0.8	8	9	9

Table 12.7: Number of unique patterns in each negative positive split of the test sets

on the pattern, but also the graph from which it is taken. It should never be the case that a pattern is both anomalous and not, in the same graph. Nothing is preventing it from being both anomalous and not in separate graphs however, and as our test sets consist of several different sampled subgraphs, with patterns sampled from them, this is possible.

Furthermore, the fact that a pattern can be both negative and positive, pokes holes in the hypothesis that our perfect performance is due to a feature inherent to just the pattern itself. It also pokes holes in the explanation that it is due to memorization. Because a given pattern can be both positive and negative, a model that relies entirely on the information contained in a pattern to classify it, whether it is some common features across all positive or negative patterns, or by memorizing it from the training sets, has no guarantee of being correct when encountering that same pattern in the test sets.

In order to show the degree to which positive patterns may also be negative in other subgraphs, and vice versa. We can run the same pattern overlap search procedure again, only this time across positive and negative classes; both from the test sets to the training sets, and within the test sets themselves. The degree of cross class pattern overlap between the test sets and the training sets is shown in Table 12.8, and the degree of cross class overlap within the test sets is shown in Table 12.9.

In Table 12.8 we can see that in most benchmarks the patterns from the positive test sets occur with a high frequency in the negative training sets. In the opposite case however, we see that only patterns from the negative test sets of the LastFM based benchmarks occur in the positive training sets.

Looking at Table 12.9, showing the cross overlap within the test sets, we can see that for around half of the bechmarks there is complete overlap of negative patterns among the positive patterns, while for the other half

k	Dataset	δ	Negative in positive train	Positive in negative train
2	IMDB	1.1	0.0	1.000
		1.2	0.0	1.000
	LUBM	0.6	0.0	0.778
		0.7	0.0	0.889
	LastFM	**0.4**	**1.0**	**1.000**
		0.7	**1.0**	**1.000**
3	IMDB	0.6	0.0	1.000
		0.625	0.0	1.000
	LUBM	0.425	0.0	0.793
		0.5	0.0	0.901
	LastFM	**0.6**	**1.0**	**0.840**
		0.8	**1.0**	**1.000**

Table 12.8: Pattern overlap between negative test patterns in the positive training class, and vice versa

k	Dataset	δ	Negative in positive test	Positive in negative test
2	IMDB	1.1	1.000	0.308
		1.2	0.000	0.000
	LUBM	0.6	0.000	0.000
		0.7	0.000	0.000
	LastFM	**0.4**	**1.000**	**0.789**
		0.7	**1.000**	**0.699**
3	IMDB	0.6	1.000	0.096
		0.625	0.000	0.000
	LUBM	0.425	1.000	0.024
		0.5	0.048	0.123
	LastFM	**0.6**	**1.000**	**0.801**
		0.8	**1.000**	**0.964**

Table 12.9: Pattern overlap between negative test patterns and positive test patterns across splits

there is no overlap. The degree of overlap of positive patterns among the negative patterns is lower however. The exception is for the LastFM based benchmarks, in those cases the overlap is significant in both directions.

12.3.3 The Confusion Hypothesis

With the pattern overlap information presented in Tables 12.8 and 12.9 a possible explanation for some of the classification performance scores appears. The perfect, and the recurring scores within benchmarks across models, may be explained by the degree of cross class overlap within the classes of a given benchmark.

As shown in Tables 12.2 and 12.3 most of our models perform better on the task of classifying negative samples. The exception being for the the k2-LastFM-δ0.4, k2-LastFM-δ0.7 and k3-LastFM-δ0.6 benchmarks. In those cases TP performance largely beats TN performance. For those benchmarks, it is also uniquely the case that all of the patterns in the negative test set, are found in the positive training set (shown in Table 12.8). If it is the case that the model relies on memorizing patterns to a degree, this could explain the model's poor accuracy on the negative test samples as the patterns in all of those samples have been seen as positive at some point during training, which is not the case with any other benchmark.

Looking at Table 12.9, the LastFM benchmarks also show a relatively large degree of overlap between the patterns in the in the positive test set in the negative test set. Whether this affects the accuracy on the negative subset or the positive is hard to say as both could be the case. The fact that the degree of this overlap is much higher with the LastFM benchmarks than with any other benchmark, however, is notable.

Chapter 13

Conclusion, Limitations and Future Work

This thesis had two goals. To provide a mathetmatically precise and implementable definition of structural anomalies in graphs. And to investigate whether a end-to-end GNN approach to anomaly detection in knowledge graphs could classify anomalies faster than a symbolic SPARQL based approach. And if so, at what cost in terms of classification performance.

In regards to the first goal we have provided a precise definition of what a structural anomaly in a knowledge graph is in Section 8.1. We have also shown that it is readily implementable using openly available semantic Web tools such as SPARQL.

In addition we created and evaluated several GNN based systems that attempt to classify structural anomalies as we've defined them. In Chapter 12 we show that the systems we created are much faster than the precise SPARQL based approach. In the worst case the GNN based system is still over 70 times faster than the SPARQL method, while being several tens of thousands times faster in the best cases.

In terms of classification performance our systems show mixed results. In some cases the systems show perfect performance (MCC of 1.0), in others the systems perform no better than a random baseline (MCC of 0.0). In most cases however, the performance lies somewhere in between, which is to be expected of an AI based approach. As such these systems may find use in real-life scenarios in which throughput rate is paramount, while lower classification performance is tolerable. In some cases the various systems perform somewhat suspiciously in that they produce identical performance scores.

In an attempt to explain the suspicious perfect and similar scores, as they are rare and often indicative of bugs in the context of machine learning, we looked at the positive and negative class performance independently. Furthermore, we investigated the number of unique patterns in our various benchmarks. In doing the latter, we show that the number of unique patterns in most of our benchmarks is quite low for the benchmarks not based on the LUBM graphs. We also show that due to an inherent property of our

training and test sample creation procedure there is a significant degree of pattern overlap between our postive and negative training and test classes. This degree of overlap in patterns found in the negative and positive classes of our training and test sets may explain why some systems struggle on particular benchmarks while performing well on others.

In order to gain further insight in the performance of our systems a suggestion for future work is to evaluate the systems on benchmarks created in a similar manner as outlined in Section 11.1 on graphs with a larger amount of node and edge types, with larger patterns. However, as the pattern size limit we imposed in this work are bounded by computational limits, creating benchmarks with patterns of a larger size would require additional resources.

As we also have significant degree of cross-class overlap due to our benchmark creation procedure (see Section 11.1). A possible avenue for future work could be the introduction of a normalizing component to our definition of structural anomalies (see Section 8.1). If we were able to normalize our measure of anomalousness based on some feature summarizing the "structure" of a graph, we might be able to train on patterns in smaller subgraphs of a larger graph, and have that training experience generalize better to patterns in the larger graph.

The fact that our systems outperform random baselines even in cases where there is significant cross-class pattern overlap, does however suggest that our systems might be learning some normalizing feature of the graphs they process. As such, another possibility for future work may be tweaking our systems such that they also process some graph-level information directly, so that they may more readily learn these possible normalizing features.

Bibliography

[1] Leman Akoglu, Hanghang Tong and Danai Koutra. 'Graph based anomaly detection and description: a survey'. In: *Data mining and knowledge discovery* 29.3 (2015), pp. 626–688.

[2] Saad Albawi, Tareq Abed Mohammed and Saad Al-Zawi. 'Understanding of a convolutional neural network'. In: *2017 international conference on engineering and technology (ICET)*. Ieee. 2017, pp. 1–6.

[3] Peter W Battaglia et al. 'Relational inductive biases, deep learning, and graph networks'. In: *arXiv preprint arXiv:1806.01261* (2018).

[4] James Bergstra et al. 'Algorithms for hyper-parameter optimization'. In: *Advances in neural information processing systems* 24 (2011).

[5] Richard J Bolton, David J Hand et al. 'Unsupervised profiling methods for fraud detection'. In: *Credit scoring and credit control VII* (2001), pp. 235–255.

[6] Lei Cai et al. 'Structural Temporal Graph Neural Networks for Anomaly Detection in Dynamic Graphs'. In: *arXiv preprint arXiv:2005.07427* (2020). URL: https://arxiv.org/pdf/2005.07427.pdf.

[7] Varun Chandola, Arindam Banerjee and Vipin Kumar. 'Anomaly detection: A survey'. In: *ACM computing surveys (CSUR)* 41.3 (2009), pp. 1–58.

[8] Nikan Chavoshi, Hossein Hamooni and Abdullah Mueen. 'Debot: Twitter bot detection via warped correlation.' In: *Icdm*. 2016, pp. 817–822.

[9] Davide Chicco and Giuseppe Jurman. 'The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation'. In: *BMC genomics* 21.1 (2020), pp. 1–13.

[10] Kyunghyun Cho et al. 'Learning phrase representations using RNN encoder-decoder for statistical machine translation'. In: *arXiv preprint arXiv:1406.1078* (2014).

[11] Kyunghyun Cho et al. 'On the properties of neural machine translation: Encoder-decoder approaches'. In: *arXiv preprint arXiv:1409.1259* (2014).

[12] Munmun De Choudhury et al. 'Social synchrony: Predicting mimicry of user actions in online social media'. In: *2009 International conference on computational science and engineering*. Vol. 4. IEEE. 2009, pp. 151–158.

[13] Qi Ding et al. 'Intrusion as (anti) social communication: characterization and detection'. In: *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2012, pp. 886–894.

[14] Alexey Dosovitskiy et al. *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale*. 2020. DOI: 10.48550/ARXIV.2010.11929. URL: https://arxiv.org/abs/2010.11929.

[15] Xinyu Fu et al. 'MAGNN: Metapath Aggregated Graph Neural Network for Heterogeneous Graph Embedding'. In: *Proceedings of The Web Conference 2020*. ACM, Apr. 2020. DOI: 10.1145/3366423.3380297. URL: https://doi.org/10.1145%2F3366423.3380297.

[16] William L. Hamilton, Rex Ying and Jure Leskovec. 'Inductive Representation Learning on Large Graphs'. In: *CoRR* abs/1706.02216 (2017). arXiv: 1706.02216. URL: http://arxiv.org/abs/1706.02216.

[17] Sepp Hochreiter and Jürgen Schmidhuber. 'Long short-term memory'. In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[18] Thomas N Kipf and Max Welling. 'Semi-supervised classification with graph convolutional networks'. In: *arXiv preprint arXiv:1609.02907* (2016). URL: https://arxiv.org/pdf/1609.02907.pdf.

[19] Mohit Kumar, Rayid Ghani and Zhu-Song Mei. 'Data mining to predict and prevent errors in health insurance claims processing'. In: *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2010, pp. 65–74.

[20] Srijan Kumar et al. 'Edge weight prediction in weighted signed networks'. In: *2016 IEEE 16th International Conference on Data Mining (ICDM)*. IEEE. 2016, pp. 221–230.

[21] *Lehigh University Benchmark (LUBM)*. http://swat.cse.lehigh.edu/projects/lubm/. Accessed: 2022-11-12.

[22] Liam Li et al. 'Massively parallel hyperparameter tuning'. In: *arXiv preprint arXiv:1810.05934* 5 (2018).

[23] Meng Liu, Hongyang Gao and Shuiwang Ji. 'Towards deeper graph neural networks'. In: *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2020, pp. 338–348. URL: https://arxiv.org/pdf/2007.09296.pdf.

[24] Ilya Loshchilov and Frank Hutter. 'Fixing Weight Decay Regularization in Adam'. In: *CoRR* abs/1711.05101 (2017). arXiv: 1711.05101. URL: http://arxiv.org/abs/1711.05101.

[25] Andrew Kachites McCallum et al. 'Automating the construction of internet portals with machine learning'. In: *Information Retrieval* 3.2 (2000), pp. 127–163.

[26] Tore Opsahl and Pietro Panzarasa. 'Clustering in weighted networks'. In: *Social networks* 31.2 (2009), pp. 155–163.

[27] Myle Ott, Claire Cardie and Jeff Hancock. 'Estimating the prevalence of deception in online review communities'. In: *Proceedings of the 21st international conference on World Wide Web*. 2012, pp. 201–210.

[28] Adam Paszke et al. 'PyTorch: An Imperative Style, High-Performance Deep Learning Library'. In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[29] Michael Schlichtkrull et al. *Modeling Relational Data with Graph Convolutional Networks*. 2017. DOI: 10.48550/ARXIV.1703.06103. URL: https://arxiv.org/abs/1703.06103.

[30] Ashish Vaswani et al. 'Attention is all you need'. In: *arXiv preprint arXiv:1706.03762* (2017). URL: https://arxiv.org/pdf/1706.03762.pdf.

[31] Petar Veličković et al. 'Graph attention networks'. In: *arXiv preprint arXiv:1710.10903* (2017). URL: https://arxiv.org/pdf/1710.10903.pdf.

[32] W3C. *World Wide Web Consortium Publishes Public Draft of Resource Description Framework (RDF)*. online. 1997. URL: https://www.w3.org/Press/RDF (visited on 28/09/2022).

[33] Xuhong Wang et al. 'One-Class Graph Neural Networks for Anomaly Detection in Attributed Networks'. In: *arXiv preprint arXiv:2002.09594* (2020). URL: https://arxiv.org/pdf/2002.09594.pdf.

[34] Li Zheng et al. 'AddGraph: Anomaly Detection in Dynamic Graph Using Attention-based Temporal GCN.' In: *IJCAI*. 2019, pp. 4419–4425. URL: https://www.ijcai.org/Proceedings/2019/0614.pdf.