Anomaly detection Techniques as a quality evaluation of graphs

by

Luca Lagunas
Proj. Supervisor: Prof. dr. Dr Yannis Velegrakis
2nd Examiner: dr. IR (Ioana) Karnstedt-Hulpus

Anomaly detection Techniques as a quality evaluation of graphs

Luca Lagunas
Proj. Supervisor: Prof. dr. Dr Yannis Velegrakis
2nd Examiner: dr. IR (Ioana) Karnstedt-Hulpus
Applied Data Science
Utrecht University
2022

Abstract

The goal of this project is the implementation of PyGQE, a software package that given a graph measures its quality by measuring the possible anomaly detections. The aim of this application is to help data scientists evaluate how important a dataset in graph form is and its level of quality. The program is implemented in python, it takes a list of edges in CSV format and a feature map (optional) and returns a list of anomalous nodes and uncommon features patterns.

Contents

Chapter 1

Introduction

The world is becoming increasingly data driven. Insight from data powers companies, universities, research, entertainment and services of any kind. There's one catch: storage is not an unlimited resource. Even with denser, faster and energy e cient memory chips, the costs and maintenance required to keep data centers running can be a game not worth the candle.

So, if the necessity of deleting a database arises for the aforementioned or other reasons, can we maybe save the most relevant information in it and save storage space by getting rid of the "bad" parts? This project tackles this very problem in regards of databases in graph form. We de ne a "bad" part of a graph as a part with anomalies, whether that be in its structure or features. We deal with these two aspects separately, implementing di erent approaches in a Python script. The project can be divided in two main parts: anomalous node detection and uncommon feature patterns. Both parts are computed in sequence once PyGQE is executed. Part one utilizes selected Deep Learning models from the PyGOD library [14] to return a ltered node list of the most probable outliers.

Part two utilizes a sequence-to-pattern generation library [**seq2pat2022**] to discover sequential patterns in the nodes' attributes that occur most (in)frequently.

Chapter 2

Related work

Anomaly detection refers to the problem of finding patterns in data that do not conform to expected behavior. To the best of our knowledge there is a relatively limited research and software diversity related to static graph anomaly detection, while many time tested techniques are available for more traditional databases forms [2]. William Eberle and Lawrence Holder [6] define three possible graph changes:

- label modifications
  - The label on a vertex is different than was expected
  - The label on an edge is different than was expected
- vertex/edge insertions
  - A vertex exists that is unexpected
  - An edge exists that is unexpected.
- vertex/edge deletions
  - An expected vertex is absent
  - An expected edge between two vertices is absent

They present three different algorithms that address these three possible changes by relying on the minimum description length principle [19], which states that the best description of the data is given by the model which compresses it the best. These algorithms determine the normative pattern as the one that minimizes the description length of a graph and then they look for patterns that deviates from it by calculating the cost of transformation (first algorithm), probability of extension to match the normative pattern (second algorithm) or look for the maximum partial substructure.

The three algorithms showed promising performances with synthetic data, scoring high detection rates with low false positive instances. Unfortunately there is no publicly available implementation.

With the increasingly efficacy and popularity of deep learning techniques new anomaly detection algorithms emerged as deep learning networks can handle the complexity of graph data very well,

the main limitation of more traditional approaches [16]. The aforementioned survey lists some of the challenges of GAD (Graph Anomaly Detection) and DGNN's (Deep Graph Neural Networks):

- Anomaly-aware training objectives:
  To nd optimal values for the model's training appropriate training objectives or a loss function is required. It is challenging to nd those for graph based data, as there is no ground-truth about the anomalies or what precisely makes them deviate from normality.
- Anomaly interpretability:
  Justifying the discovered anomalies is one of the biggest downsides of DL methods and it can be problematic in some scenarios (e.g. blocking a bank account for anomalous use must be supported by evidence)
- High training cost:
  These DGNN's are capable of handling both structural data and attributes, making them particularly more complex than traditional networks. This complexity translates into increased time and computational costs.
- Hyperparameter tuning:
  Like for most neural networks the outcome of these models is heavily in uenced by their hyperparameters, so ne-tuning the values for the hyperparameters is a must to successfully perform at the task at hand. The problem with unsupervised graph anomaly detection arises because we don't have labeled data to judge the model.

Despite these shortcomings graph anomaly detection is considered to be at the forefront of anomaly detection and [16] proposes 12 future directions to address and overcome the listed problems. The aim of this thesis project is not to implement any of these suggestions nor to justify the outcome of the DL models utilized in the program.

The library we utilized is PyGod [14], a Python library for graph outlier detection. It's developed on top of PyTorch Geometric and PyTorch and it o ers several anomaly detection algorithms. Here's the list of the one that can be used in PyGQE:

- MLPAE [20]
- GCNAE [12]
- DOMINANT [5]
- DONE [1]
- AdONE [1]
- AnomalyDAE [8]
- GAAN [3]
- CoLA [15]
- ANEMONE [11]

- GUIDE [23]

For the second part of the project we find the uncommon feature patterns in the graph by leveraging the Seq2Pat library [**seq2pat2022**]. We don't know of existing research that uses sequential patterns discovery algorithms to look for anomalies.

Chapter 3

Implementation

In this chapter we are going to present the key aspects of PyGQE's implementation, from the chosen libraries to the output, showing snippets of relevant code.

3.1 Libraries

Given the nature and the time constraints of this project we had to rely on existing libraries for many tasks rather that implementing our own solutions. Luckily all of them o ered enough exibility and functionalities to get the results we wanted. Here's the list of all the PyGQE imports:

```
import click
import pandas as pd
import numpy as np
import networkx as nx
import torch
import walker
from torch_geometric.data import Data
from pygod.models import MLPAE
from pygod.models import GCNAE
from pygod.models import DOMINANT
from pygod.models import DONE
from pygod.models import AdONE
from pygod.models import AnomalyDAE
from pygod.models import GAAN
from pygod.models import CoLA
from pygod.models import ANEMONE
from pygod.models import GUIDE
from numpy import array_equal
from sequential.seq2pat import Seq2Pat
from pyvis.network import Network
```

3.1.1 Click

Click [4] is a Python package for creating modular command line interfaces with as little code as necessary. Click allows the creation of both arguments and optional parameters. Arguments can be mandatory and are always positional, meaning that if a default value is not specified in the code then the user must provide the data and it must be presented in the required positional order. Both arguments and options can be statically typed and have a default value. Options can be declared having two names: a full name prefixed by two dashed and a short version prefixed by a single dash. E.g. in PyGQE the threshold valued can be specified with either {threshold or -t, followed by the value.

3.1.2 Pandas and numpy

Two libraries that need no introduction, we used pandas to handle the csv inputs and for internal data organization.
Numpy was used to convert arrays and matrices into the appropriate format and type to train the models and perform efficient operations.

3.1.3 Torch and Torch Geometric (PyG)

Torch [17] and PyG [9] are the the two main packages upon which the PyGOD library [**PyGOD**] was built on. They allow tensor computation (like NumPy) with strong GPU acceleration and the creation of deep neural networks and Graph Neural Networks (GNNs).

3.1.4 walker

Walker is a[10]open-source python package that performs random walks of a specified number of steps on a graph. It's written in C++ and use parallelization for speed and computational efficency. It can indeed generate millions of random walks in a few seconds.

3.1.5 Seq2Pat

This package [21] is an implementation of the Seq2Path algorithm [**seq2pat2022**]. We leverage its basic pattern mining capabilities only, but the library allows for constraint-based sequential pattern mining and dichotomic pattern mining. The choice of this particular pattern mining algorithm and library mainly comes from an availability perspective: several other algorithms have been considered (FAST, PrefixSPAN, A priori frequent item sets etc.) but their implementation where either absent or under developed/with missing functionalities. This Seq2Path library does not require additional external inputs (some other required .txt files to operate) and can work directly with the internal data structures from the previous steps. Additionally it does not require tweaking nor additional code to work with string attributes. For the purposes of this project we expect comparable if not identical results for the mining of infrequent patterns.

3.1.6 pyvis

Pyvis [18] is a simple network visualization library. The visualization is dynamic and it's drawn in an html canvas. The user can interact woth the nodes (drag, zoom) and set several physics simulation parameters through a visual interface (sliders and selectors).

3.1.7 PyGOD

The library that makes the first step, anomalous node detection, possible. As of right now it's not possible to tweak the DL models' hyperparameters. See the Future work chapter for more.

3.2 Input

3.2.1 External inputs

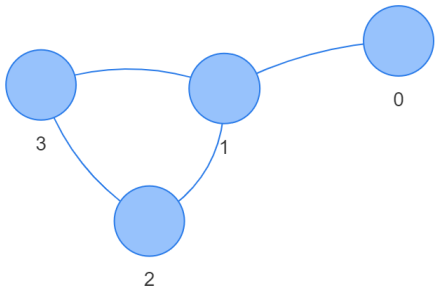
The only mandatory argument to be passed to PyGQE is the path of the CSV file that contains the list of edges of the graph. The requirements for this file are the following:

- No headers: the edge list must start from the first row
- The first column must contain the source nodes
- The first column must contain the target nodes
- Minimum node degree = 1

Regarding the last point: as of the moment PyGQE does not support rogue nodes (nodes with degree 0, no edges from or to them). If needed be it is possible to include them in the list of edges by giving them a loop, i.e. an edge with the node itself as both source and target. Here's an example of a valid .CSV edge list:

0	1
1	2
1	3
2	3

Which translates to the following graph:



The user can also specify the location of the feature map. The feature map has also to be in CSV format and it follows similar rules to the list of edges:

- No headers: the features list must start from the first row
- The first column must contain the node ids
- The first column must contain the attribute value
- All nodes must be listed. Order not important

As of the moment PyGQE considers only one node attribute when looking for infrequent patterns. See the Future work chapter for more.

3.2.2 Internal inputs

The main internal data structures and inputs given to functions are mainly lists, pandas dataframes and numpy arrays/matrices. The most complex piece of internal input and the one that requires more elaboration is the data that's fed to the DL models of PyGOD. Here's a very simple example from the PyG documentation:

```
edge_index = torch.tensor([[0, 1],
                           [1, 0],
                           [1, 2],
                           [2, 1]], dtype=torch.long)
x = torch.tensor([[-1], [0], [1]], dtype=torch.float)

data = Data(x=x, edge_index=edge_index.t().contiguous())
```

The available models in PyGQE all require the data in this same format. Because of the nature of torch tensors and how the operations are computed during the model's training it's not possible to maintain the original node IDs. Therefore the IDs are temporarily mapped to sequential integers and later converted back once the output is displayed to the user.

3.3 Parameters & execution

The user can specify seven optional parameters to control PyGQE's behaviour:

- Model
  Called through: '-m' or '{model'
  Default value: ["GUIDE"]
  Defines the model(s) used to look for anomalous nodes. Multiple values can be provided, and the results will be averaged into one.
- Number of iterations Called through: '-i' or '{iter'
  Type: int Default value: 100
  Each model is fitted to the data i times to account for slight differences in different training, making the results converge to a definite set of node outlier probabilities.
- Threshold Called through: '-t' or '{threshold'
  Type: float Default value: 0.7

The models predicts the probability of a node starting from its (uni ed) outlier score [13]. The threshold parameter sets a line on this probability over which a node is considered an outlier.

- Number of walks Called through: '-nw' or '{nwalks'
  Type: int Default value: 1000
  How many random walks should be done.

- Length of walks Called through: '-lw' or '{lwalks'
  Type: int Default value: 10
  The length of every random walk

- Number of infrequent patterns Called through: '-u' or '{npatterns'
  Type: int Default value: 5
  How many infrequent patterns to be shown. Ordered in increasing frequency.

- Visualise graph Called through: '-v' or '{viz'
  Type: bool Default value: False/F
  Whether the interactive graph showing the anomalous nodes should be rendered.

The following is a description of the logical steps PyGQE takes, from input to output:

```
1. Read the passed argument and options. Use default values if not specified
### Part 1: outlier nodes ###
2. Read the edge list and extract the list of nodes from it
3. if feature map is not passed as an argument:
       create a feature map array (torch tensor) with all values =1
   else
       create feature map array (torch tensor) with the values extracted from the
           user-provided file
4. Create maps between the original node IDs and the new progressive IDs. Create
   a new edge list torch tensor with the updated nodes' names
5. Create TORCH_GEOMETRIC.DATA object using the previously created torch tensors
6. Create a list of the models selected by the user. Fit every model a number of
   times as specified by the --iter parameter. Store the prediction values of
   every iteration in a different array per model and then average them. Average
   the results across all selected models
7. Create dataframe containing all the nodes and their predicted outlier
   probability
   Apply the threshold and print only the nodes that exceed it
### Part 2: infrequent feature patterns ###
8. if no feature map is passed as an argument:
       go to 12.
   else: go to 9.
9. Create a networkx graph object using the provided edge list and perform n
   random walks of length l as specified by the --nwalks and --lwalks parameters
   respectively
10. Replace the node IDs in the random walks with their feature value
```

```
11. Perform Seq2Path and print the n most infrequent patterns as specified by the
    --npatterns parameter
12. if the --viz parameter == True:
        create Pyvis Network object, insert nodes,edges and label utilizing
        internal data, color the nodes according to the outlier probability
        set up the buttons and interface
        plot the graph
```

3.4 Output

From a valid command line like the following: Which translates to the following graph:

Figure 3.1: edge list and feature map provided, GUIDE model selected, threshold set at 0.65, 50 iterations, random walks of length 5, top 10 most infrequent feature patterns visualized, graph visualization

```
> py .\pygqe.py .\data\edgelist.csv .\data\fmap.csv -m GUIDE -t 0.65 -i 50 -lw 5 -u 10 -v T
```

We get

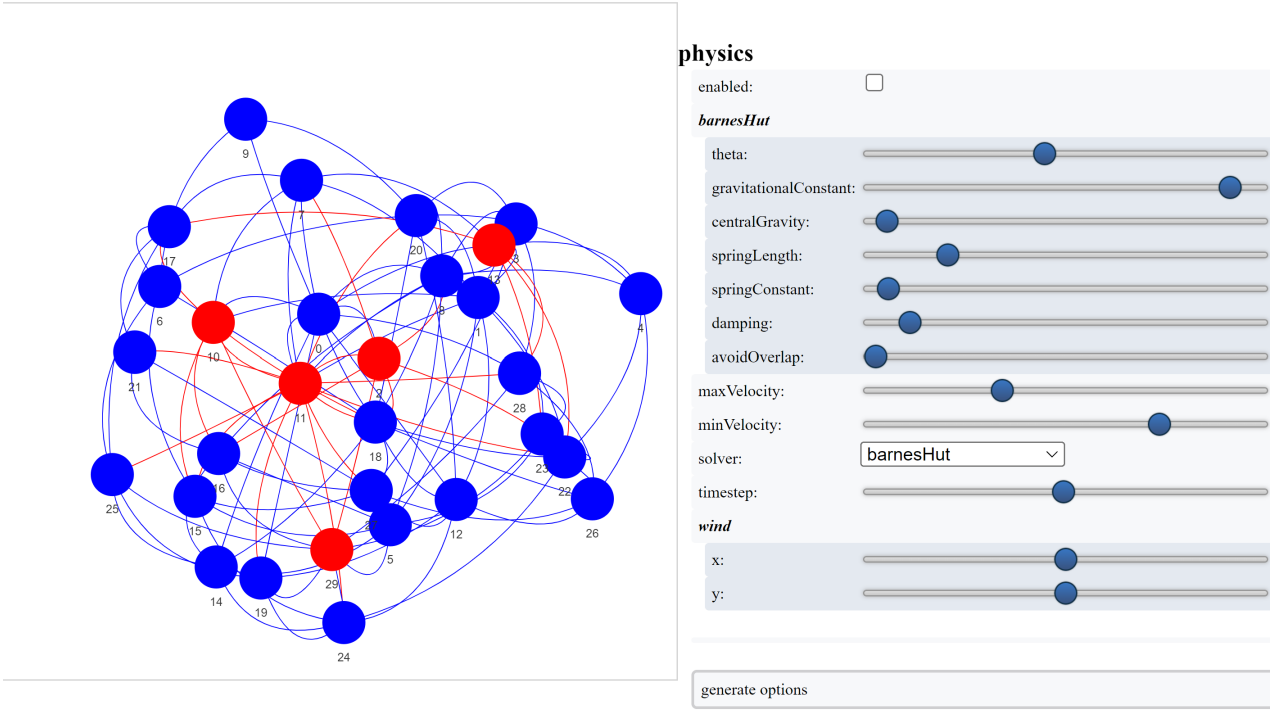
Figure 3.2: List of most anomalous nodes

```
------------
    node  outlier_probability
1      2             0.762690
5     10             0.759438
6     11             0.962985
8     13             0.808384
20    29             0.852325
```

Figure 3.3: List of most infrequent patterns

```
10 most unfrequent feature patterns
[982, 982, 982]
[982, 982, 407, 982]
[982, 407, 982, 982]
[982, 407, 982, 407, 982]
[982, 407, 407, 982]
[969, 969, 741, 969]
[969, 969, 305, 29]
[969, 969, 93, 969]
[969, 969, 47, 93]
[969, 969, 47, 29]
```

Figure 3.4: Graph visualization



Chapter 4

Tests & Conclusions

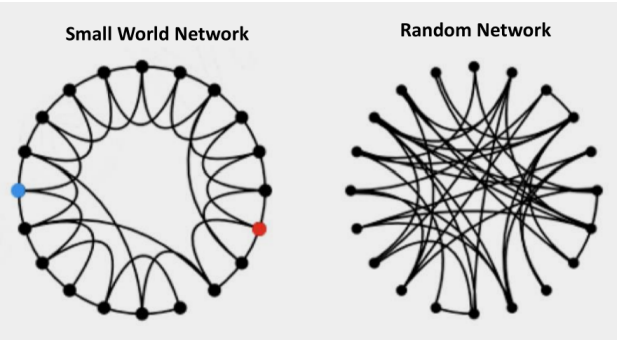
PyGQE is developed entirely in Python within VSCode and on a Windows 11 machine. It was tested both in Windows 11 and Linux Ubuntu 22.04 LTS.

We tested the script with two main sources of data: randomly generated graphs and feature maps and manually built ones.

4.1 Random graphs

The random graphs structure was generated using either the Erdős–Renyi model [7] or the Watts-Strogatz small-world model [22].
In a Erdős–Renyi model all edges have the same probability p to exist.
A Watts-Strogatz small-world graph is constructed starting from a regular ring lattice and rewiring every edge with probability p.

Figure 4.1: [22]



On a more practical point of view the two types of graphs where created using networkx graph generators (erdos_renyi_graph() and watts_strogatz_graph). We choose these two types of models fro the following reasons:

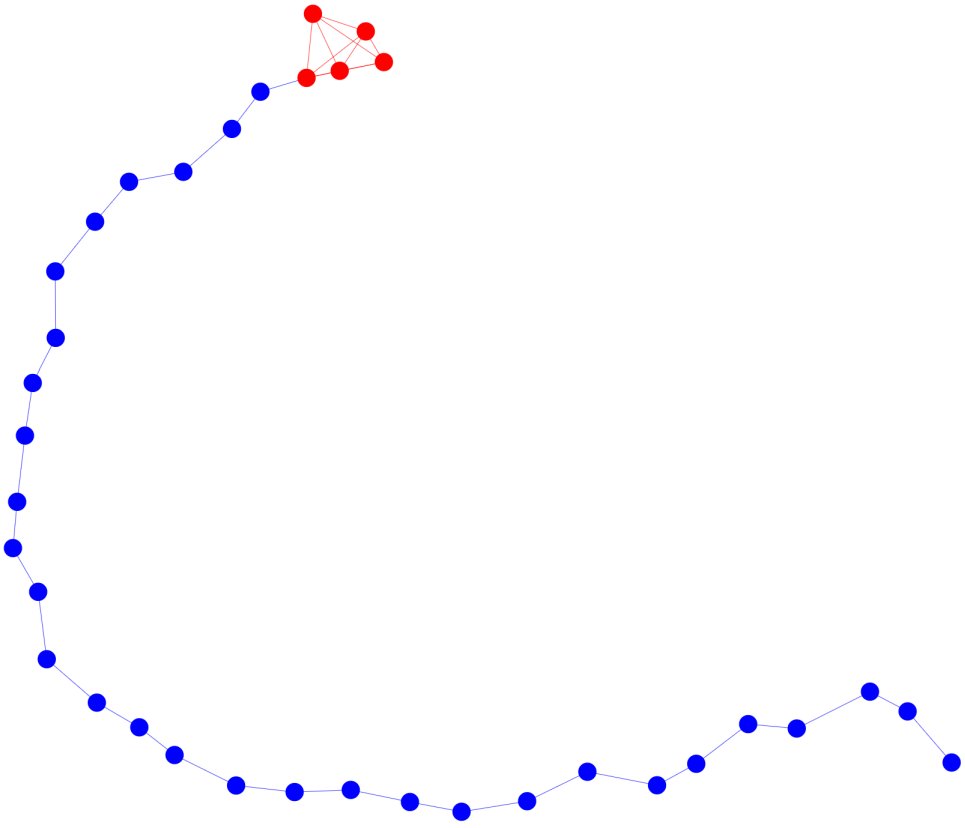
- Randomness: truly random networks to prevent biases in their creation
- Low diameter: both models have low diameters (the longest of the shortest path between two

nodes), which characterizes the ability of two nodes to communicate, meaning that the smaller the diameter the shorter the expected path between them. This increased connectivity, and therefore redundancy,is bene cial when looking for frequent patterns. More sparse networks would need more random walks to reinforce a pattern

- computational load: less random walks means reduced computational load

4.2 Manually generated graphs

We created some test graphs of small dimensions with manually injected outliers to check PyGQE behaviour. In the following example, we tested the performance of the GUIDE model (PyGQE default model) in a linear graph with an anomalous hyper connected structure on one end.



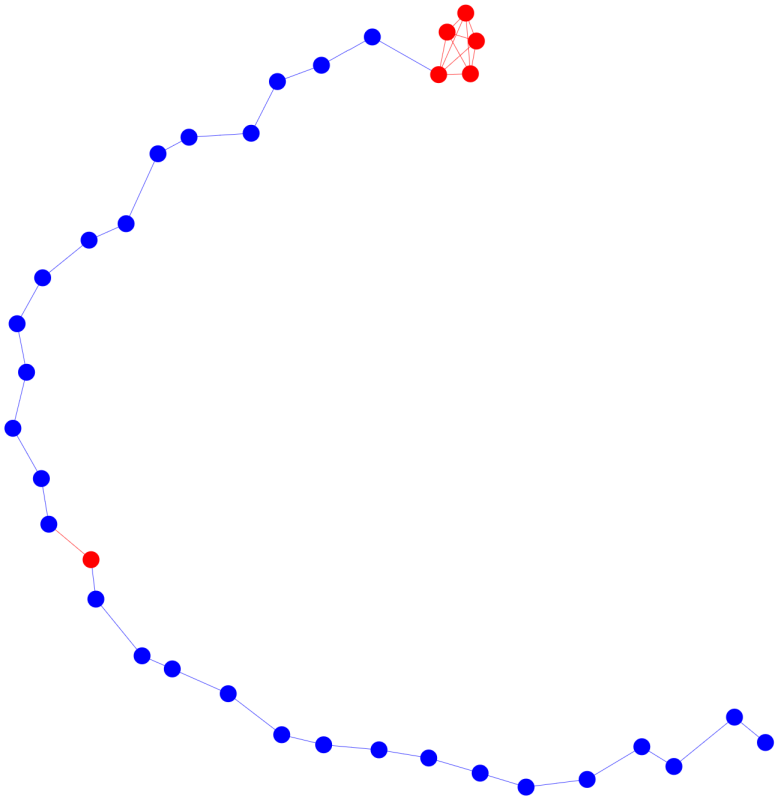
As shown in the above gure, the model correctly recognises the connected structure as anomalous with a high degree of con dence, as the probabilities below show:

The attributes of all nodes in this examples have value 1, as no feature map was provided. The anomalies are therefore purely structural.

We then manually altered the value of a random node that does not belong to the connected structure to a very high value. The model once again recognized the anomalous connected structure

```
------------
    node  outlier_probability
30    30             0.988017
31    31             0.833779
32    32             0.828910
33    33             0.826296
34    34             0.824383
```

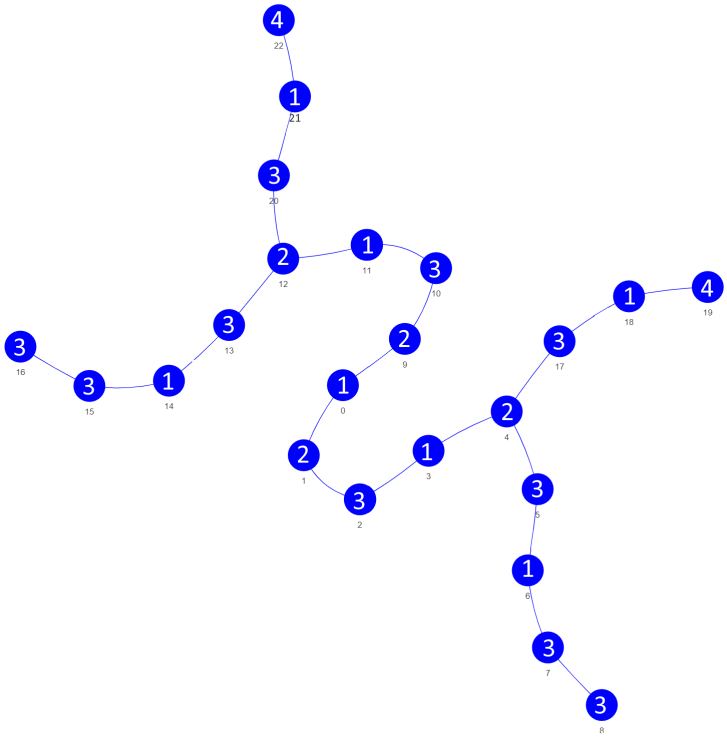
plus the altered node, as the two gures below illustrate:



Another example of manually built graph used for testing was the following:

We used this graph as one of the test for the infrequent pattern detection part. The above graph has 22 nodes and each node has an attribute value between 1 and 4 (number inside the node). The intended non anomalous progression is 1-2-3, meaning that if we are on a node of value 1 we expect the next one to be a 2 and from there to nd an adjacent node of value 3. A progression like 3-1-4

```
------------
    node  outlier_probability
15    15             0.999871
30    30             0.965106
31    31             0.861664
32    32             0.862968
33    33             0.863228
34    34             0.863203
```



or 1-3-3 would be considered anomalous, as it is an uncommon pattern trough out the graph.
The top 5 infrequent patterns found by the seq2path algorithm, after 10000 walks of length 3 are:

```
5 Most unfrequent feature patterns
[2, 1, 3]
[3, 1, 2]
[3, 4]
[3, 1, 4]
[1, 3, 3]
```

4.3 Conclusions

PyGQE is a tool to detect structural, attribute and pattern anomalies. As such it has no prior assumptions on the nature of the dataset and does not try to explain the origin nor the reason of the existence of outliers. As a tool it gives the functionalities and exibility to detect such anomalies, but it's up to the user must have both knowledge of what the database represents and of what model, hyperparameters and options to use.
The program performed well under our test conditions and the anomalies were correctly identi ed in our tests, although it is only possible to con rm that in our manually generated graphs. As

mentioned in the introduction one of the main challenges with outlier detection in graphs is the di culty to check for the correctness of the result. See the Future work chapter for an additional assessment on the matter.

We can't fully comment on the performance of PyGQE on large or very large graphs (more than 1M nodes), but computational e ciency was not the main priority of the project

Chapter 5

Future work

As a nine weeks long project there was only time to develop the main functionalities, but here's a list of possible future implementations and added functionalities to address some of the current limitations.

- Include model evaluation with labeled data: the PyGOD library o ers some utility function to evaluate models' performance when a labeled dataset is provided, i.e. labels that indicate if a node is an outlier or not
- Include possibility to ne tune models' hyperparameters: as mentioned before as of right now it is not possible to set the selected model hyperparameters. This was skipped to avoid excessive verbosity in the command line. One possible direction to overcome this is to use a txt/JSON con guration le and pass its path as an argument
- Implement support for multiple features to be analyzed: the current version of PyGQE only supports pattern mining for one attribute per node. The infrequent pattern extraction could be done on all or a selected subset of all the node's attributes
- Custom save directory and name for graph: adding the option to save the plotted graph as an HTML le with a custom name and on a user-speci ed directory
- Optimize computational e ciency: even though Python is not the fastest language around, additional computational e ciency can be achieved by optimizing the numerous loops and conversions by using more e cient methods, such as a more deliberate use of numpy (written in C++) functions.

- - - - -

Bibliography

[1] Sambaran Bandyopadhyay et al. \Outlier Resistant Unsupervised Deep Architectures for Attributed Network Embedding". In: *Proceedings of the 13th International Conference on Web Search and Data Mining*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 25{33. ISBN: 9781450368223. URL: https://doi.org/10.1145/3336191.3371788.

[2] Varun Chandola, Arindam Banerjee, and Vipin Kumar. \Anomaly Detection: A Survey". In: *ACM Comput. Surv.* 41.3 (July 2009). ISSN: 0360-0300. DOI: 10.1145/1541880.1541882. URL: https://doi.org/10.1145/1541880.1541882.

[3] Zhenxing Chen et al. \Generative Adversarial Attributed Network Anomaly Detection". In: *Proceedings of the 29th ACM International Conference on Information Knowledge Management*. CIKM '20. Virtual Event, Ireland: Association for Computing Machinery, 2020, pp. 1989{1992. ISBN: 9781450368599. DOI: 10.1145/3340531.3412070. URL: https://doi.org/10.1145/3340531.3412070.

[4] *Click*. https://github.com/pallets/click.

[5] Kaize Ding et al. \Deep Anomaly Detection on Attributed Networks". In: *Proceedings of the 2019 SIAM International Conference on Data Mining (SDM)*, pp. 594{602. DOI: 10.1137/1.9781611975673.67. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781611975673.67. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781611975673.67.

[6] William Eberle and Lawrence Holder. \Anomaly detection in data represented as graphs". In: *Intell. Data Anal.* 11 (Nov. 2007), pp. 663{689. DOI: 10.3233/IDA-2007-11606.

[7] P. Erdos and A. Renyi. In: ().

[8] Haoyi Fan, Fengbin Zhang, and Zuoyong Li. *AnomalyDAE: Dual autoencoder for anomaly detection on attributed networks*. 2020. DOI: 10.48550/ARXIV.2002.03665. URL: https://arxiv.org/abs/2002.03665.

[9] Matthias Fey and Jan Eric Lenssen. *Fast Graph Representation Learning with PyTorch Geometric*. May 2019. URL: https://github.com/pyg-team/pytorch_geometric.

[10] *graph-walker*. https://github.com/kerighan/graph-walker.

[11] Ming Jin et al. \ANEMONE: Graph Anomaly Detection with Multi-Scale Contrastive Learning". In: *Proceedings of the 30th ACM International Conference on Information Knowledge Management*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 3122{3126. ISBN: 9781450384469. URL: https://doi.org/10.1145/3459637.3482057.

[12] Thomas N. Kipf and Max Welling. *Variational Graph Auto-Encoders*. 2016. DOI: 10.48550/ARXIV.1611.07308. URL: https://arxiv.org/abs/1611.07308.

[13] Hans-Peter Kriegel et al. \Interpreting and Unifying Outlier Scores". In: *Proceedings of the 2011 SIAM International Conference on Data Mining (SDM)*, pp. 13{24. DOI: 10.1137/1.9781611972818.2. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781611972818.2. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781611972818.2.

[14] Kay Liu et al. \Benchmarking Node Outlier Detection on Graphs". In: *arXiv preprint arXiv:2206.10071* (2022).

[15] Yixin Liu et al. \Anomaly Detection on Attributed Networks via Contrastive Self-Supervised Learning". In: *IEEE Transactions on Neural Networks and Learning Systems* 33.6 (2022), pp. 2378{2392. DOI: 10.1109/TNNLS.2021.3068344.

[16] Xiaoxiao Ma et al. \A Comprehensive Survey on Graph Anomaly Detection with Deep Learning". In: *arXiv e-prints*, arXiv:2106.07178 (June 2021), arXiv:2106.07178. arXiv: 2106.07178 [cs.LG].

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

[18] *pyvis*. https://github.com/WestHealth/pyvis.

[19] Matthew J. Rattigan and David Jensen. \The Case for Anomalous Link Discovery". In: *SIGKDD Explor. Newsl.* 7.2 (Dec. 2005), pp. 41{47. ISSN: 1931-0145. DOI: 10.1145/1117454.1117460. URL: https://doi.org/10.1145/1117454.1117460.

[20] Mayu Sakurada and Takehisa Yairi. \Anomaly Detection Using Autoencoders with Nonlinear Dimensionality Reduction". In: *Proceedings of the MLSDA 2014 2nd Workshop on Machine Learning for Sensory Data Analysis*. MLSDA'14. Gold Coast, Australia QLD, Australia: Association for Computing Machinery, 2014, pp. 4{11. ISBN: 9781450331593. DOI: 10.1145/2689746.2689747. URL: https://doi.org/10.1145/2689746.2689747.

[21] *seq2pat*. https://github.com/fidelity/seq2pat.

[22] Duncan J. Watts and Steven H. Strogatz. \Collective dynamics of `small-world' networks". In: 393.6684 (June 1998), pp. 440{442. DOI: 10.1038/30918.

[23] Xu Yuan et al. \Higher-order Structure Based Anomaly Detection on Attributed Networks". In: *2021 IEEE International Conference on Big Data (Big Data)*. 2021, pp. 2691{2700. DOI: 10.1109/BigData52589.2021.9671990.