Joaquín Cabezas

Improving stability of GNNExplainer in large citation network datasets

Master of Sciences Final Project

directed by Dr. Jordi Duch Gavaldà

Master's Degree in Computer Security Engineering and Artificial Intelligence

UNIVERSITAT
ROVIRA i VIRGILI

Tarragona

2021

Abstract

Graph Neural Networks (GNNs) is a Machine Learning framework that brings neural networks to graph and relational data. It is of special relevance for areas like social network analysis, biological sciences, chemistry, smart transportation systems and many others, where data can be thought of as a network. Explaining why a GNN made a decision is a challenge, due to the black-box nature of neural networks, but it is crucial when applying it to decision-making processes that affects the life of many. In this work we review the current state of the art and analyze the most well-known method for explaining GNNs, GNNExplainer. We find that its application to academic citations datasets present issues due to the variability of the explanations and we propose a modification for improving stability of the results and interpretability of the graphical explanation. In particular, we propose the use of an adjusted coefficient computed beforehand for every explanation instead of a fixed parameter. We find that our proposal improves the stability by more than 10 % in experiments using Graph Convolutional Networks (GCN) and Graph Attention Networks (GAT) with two citation networks datasets (Cora and Pubmed).

Contents

1 Introduction **6**
- 1.1 Motivation . . . 6
- 1.2 Scope . . . 7
- 1.3 Research questions . . . 7
- 1.4 Outline . . . 7

2 Background **8**
- 2.1 Machine Learning . . . 8
- 2.2 Graph Machine Learning . . . 8
- 2.3 Explainability for Graph Machine Learning . . . 11
- 2.4 State of the art . . . 12
  - 2.4.1 GNNExplainer . . . 14
  - 2.4.2 PG-Explainer . . . 15
  - 2.4.3 COGE . . . 16
  - 2.4.4 PGM-Explainer . . . 17
  - 2.4.5 GraphLIME . . . 18
- 2.5 Computer methods . . . 19
- 2.6 Datasets . . . 19

3 Explainability of GNNs **21**
- 3.1 Problem statement . . . 21
- 3.2 Need for heuristics . . . 22
  - 3.2.1 Determining combinations within the computation graph . . . 22
  - 3.2.2 Reducing the search space . . . 23
  - 3.2.3 Quantifying the need of heuristics in citation network datasets . . . 25
- 3.3 Need for stability . . . 26

4 Understanding and improving explainability of GNNExplainer **28**
- 4.1 Effect of randomness in GNNExplainer . . . 28
- 4.2 Effect of the number of edges in the loss function . . . 31
- 4.3 Adjusting the edge_size coefficient . . . 36
- 4.4 Using Monte Carlo to reduce instability . . . 38
- 4.5 Benchmarks of stability . . . 39
- 4.6 Improving the visualization of GNNExplainer . . . 41

5 Conclusions **46**
- 5.1 Drawbacks . . . 47
- 5.2 Future work . . . 48

A Reproducibility instructions **49**

B Node sampling **51**

Bibliography **56**

List of Figures

2.1 Node embeddings . . . 9
2.2 Overview of the encoder-decoder approach . . . 9
2.3 Layers of a Graph Neural Network . . . 10
2.4 Graph Attention Networks . . . 11
2.5 Taxonomy of explainers for GNNs . . . 12
2.6 Conceptual idea of GNNExplainer . . . 14
2.7 Flow of PG-Explainer . . . 16
2.8 Architecture of PGM-Explainer . . . 17
2.9 Conceptual flow of GraphLIME . . . 18

3.1 Search space for a rooted binary tree of two levels . . . 24
3.2 Two consecutive executions for explaining node 10 at Cora Dataset . . . 26
3.3 Two consecutive executions for explaining node 10 at Pubmed Dataset . . . 27

4.1 Explanation for node 51 at Cora dataset . . . 29
4.2 Evolution of edge mask value during explanation (Cora dataset) . . . 30
4.3 Evolution of edge mask value during explanation (Cora dataset) . . . 30
4.4 Proportion of bistable behaviour in edges (Cora dataset) . . . 31
4.5 Contribution to the loss function for node 10 at Cora . . . 33
4.6 Contribution to the loss function for node 112 at Cora . . . 33
4.7 Distribution of edge mask value for node 112 at Cora . . . 34
4.8 Distribution of N in the synthetic dataset number 1 from the original article . . . 34
4.9 Distribution of N in the synthetic dataset number 2 from the original article . . . 35
4.10 Distribution of N in the Cora dataset . . . 35
4.11 Distribution of N in the Pubmed dataset . . . 36
4.12 Proportion of bistable behaviour in edges after coefficient adjustment (Cora dataset) 37
4.14 2-dimensional array of edge mask values for node 10 at Cora Dataset . . . 38
4.13 Explanation for node 51 at Cora Dataset. Left: Default Settings. Right: Adjusted Coefficient . . . 38
4.15 Comparison of the proportion of bistable behaviour in edges . . . 41
4.16 Default explanation for node 112 at Cora with default settings . . . 42
4.17 Our proposed explanation for node 112 at Cora . . . 42
4.18 Default explanation for node 7346 at Pubmed . . . 43
4.19 Our proposed explanation for node 7346 at Pubmed . . . 43
4.20 Explanation for node 1560 at Cora Dataset. Left: Default visualization. Right: Proposed visualization . . . 45

B.1 Comparison of the histograms of egocentric networks . . . 52

List of Tables

4.1 Different policies selecting relevant edges 39
4.2 Evaluation of stability 40
4.3 Evaluation of proportion of white pixels 45

Listings

4.1 Code for mask setting in GNNExplainer 28
4.2 Coefficients in GNNExplainer 32
4.3 Code for our modified loss function in GNNExplainer 32
4.4 Code for calling the explainer 37
4.5 Code for calling the explainer with Monte Carlo method 39
4.6 Graphical explanation function 43
4.7 Code for preparing the explanation sub-graph 44
B.1 Code for obtaining the number of edges in the k-hop egocentric network 51
B.2 Code for DegreeBasedSampler in LittleBallOFFur 52

Chapter 1

Introduction

1.1 Motivation

One of the major subsets of Artificial Intelligence is Machine Learning. It is quite revolutionary on our times as it yields impressive results, in many occasions outperforming humans for some narrow tasks as detecting patterns in images or translating texts to any language. It is essentially based on building a model based on past sample data, in order to make predictions (filling in missing information). Instead of just looking for past examples and presenting the closest match, Machine Learning is able to find patterns and, through induction, infer an outcome for new unseen examples.

If we focus on the data, we may imagine a spreadsheet-alike table, where every row is an example and the columns represent the properties of that example (or features, as it is named in the field). Following this type of structure, called tabular, we have examples on prediction for house prices, industrial maintenance, insurance prediction and so on. Similarly, images can be understood as structured; just imagine that every pixel in an image is a cell in your spreadsheet, with a certain amount of red, green and blue. This data structure is informally called Euclidean, as in the Euclidean geometry, because there is the intuition of a grid-like structure that can be defined in Rn, in opposition to data within graphs (for example trees and other hierarchies).

Many applications need to use data in the form of graphs, for example social networks, molecular chemistry, physics systems and many more. For these applications, applying the Machine learning methods without taking into account the relational information lead to poor results. To leverage this information, a new kind of methods arise, like Graph Machine Learning, Geometric Deep Learning or Graph Neural Networks.

These methods typically use message passing between the neighbors to update its features. To limit the computation cost, the neighbors for each node are limited to a number of hops of distance, called a k-hop neighborhood. The standard operations conducted are aggregating the features of the neighbors and combining them with the features of the node itself. After some repetitions of this procedure, the answer is ready.

As in any Neural Network, after the training there is a black-box situation, where the model works but it is not possible to explain why. This is a major problem for many users with legal constraints or moral issues. As the Machine Learning applications spread and its use became pervasive, so the explainability or interpretability became critical [1].

We will explore the current methodologies for explaining (single-instance) predictions for node classification under Graph Neural Networks, and will propose a method for setting the explainer parameters automatically, while providing experimentation with classical benchmarking datasets.

1.2 Scope

This work is focused on the Explainability for Graph Neural Networks, posed as the attribution problem, which consist on identifying the parts of an input that are responsible for the prediction. We will conduct a study on the state of the art and will present related mathematical methods and computational techniques.

In particular, we will choose GNNExplainer as our tool for explanation, we will test it with two popular datasets of citation networks and we will propose a method to reduce instability.

Finally, we will explore different visualization techniques for presenting the explanations.

1.3 Research questions

This work will try to tackle the two following research questions:

Question 1. Do the tuning parameters of a model impact the explanation? Our hypothesis is that a fixed parameter will only work a narrow range of distributions of node's neighborhoods and will produce unstable explanations for the nodes outside that range. If this is the case, how can we set the parameters to reduce this issue?

Question 2. Does the random initialization mask impact the explanation? Our hypothesis is that results are highly dependent on the pseudo-random number generator. If it depends on the initialization mask, we will assess which strategy should we follow and if it needs to be adjusted at the model, dataset and/or node level.

1.4 Outline

In chapter 2 we provide a background on the major topics of this report, paving the way for the analysis of the current state of the art, where we will describe the methods for those articles with an open source implementation. In chapter 3 we present the general problem an we introduce the need of heuristics and stability. In chapter 4 we choose GNNExplainer as our tool for explanation and we present the results of our proposed modification for improving stability and we discuss a new visualization function for plotting explanations. In chapter 5 we summarize our work, highlight the achievements and we discuss future work.

Chapter 2

Background

In this chapter we will present each topic with the basic introduction and will present the terminology and notation. After these short introductions, we will present the problem of explainability for Graph Neural Networks.

2.1 Machine Learning

Machine learning is a set of methods to make predictions based on data. It represents a paradigm shift from classical programming where all instructions must be explicitly given to the computer, to another strategy where the instructions are not provided. There are different approaches depending on the scenario:

- Supervised machine learning is related to prediction problems where we have a dataset for which we already know the outcome of interest.
- Unsupervised learning do not have a specific outcome of interest, but we find patterns or clusters of data points with similar properties.
- Reinforcement learning, where an agent learns to optimize a certain reward by acting in an environment.

In this work we focus on supervised machine learning, particularly at the classification tasks where we aim to identify to which category an unseen observation belongs. This ability to predict unseen observation is called induction, and it refers to learning a function that can be applied to novel inputs by means of general rules, in opposition to transductive, where we look for observed examples to rely on them.

2.2 Graph Machine Learning

In Graph Machine Learning the input data provides not only the features of each node, but also the relationship between the nodes so we can model the data as a graph. In a graph there is a collection of vertices or nodes holding some data about themselves and about how they are connected to other nodes (typically this connection is called edge or link). It can serve as an abstraction of a network. In this work we are interested in examples where the graph represent interactions between humans, particularly example citations on an academic network, but this technique can be applied to social networks analysis [2], medical and biological sciences [3] and many others.

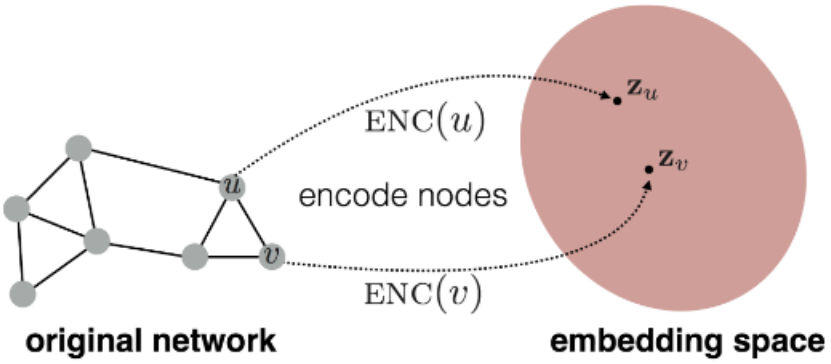

Figure 2.1: Node embedding, obtained from [5]. The concept of distance in the latent space is related to the relationships of the original graph

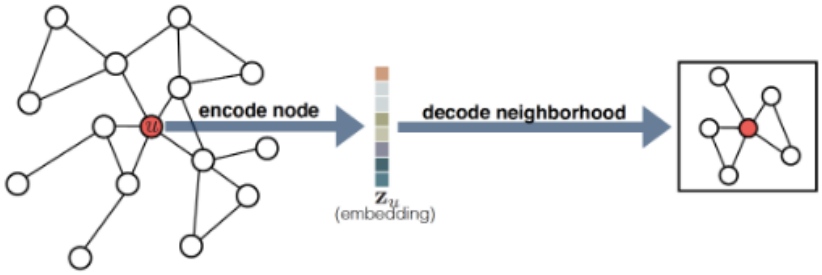

Figure 2.2: Overview of the encoder-decoder approach, obtained from [5]

Traditional techniques of Machine Learning can not be applied directly to this data because the relational information will be lost. Instead, we have to look at adaptations or radical new methods. This does not mean that we can not rely on the foundations of Machine Learning, it is only that the algorithms will be somehow different. The basics of supervised or unsupervised machine learning are the same, and so are the optimization techniques and many other aspects. There are, of course, new tasks like node classification or link prediction, but we can find similarities with classical Machine Learning tasks.

Graph Machine Learning is also called Geometric Deep Learning, which as stated in [4] is an umbrella term for methods trying to generalize deep neural models to non-Euclidean domains such as graphs and manifolds. The concept of non-Euclidean domain is used to denote the challenges arising by the lack of some properties found in grids of data like images or tabular data. For example, the convolution operation is well defined in a 2D image where all pixels are distributed in a regular lattice, but not for an arbitrary graph.

Among the many approaches to Graph Machine Learning we are interested in those within the Representation Learning paradigm. Representation learning consists on encoding network structure into low-dimensional embeddings, typically using techniques of deep learning and nonlinear dimensionality reduction. This low-dimensional embeddings are vectors in a latent space where the geometric relations correspond to the relationships of the original graph, as show in Figure 2.1.

The idea of using embeddings is related to the encode-decode architecture, where we have an encoder functions that maps nodes $v \epsilon V$ to vector embeddings $z_v \epsilon \mathbb{R}^d$ and a decoder that can provide graph statistics about the vector embedding, as depicted in Figure 2.2. As summarized in [6], there are many different methods leveraging the graph representation learning: DeepWalk [7] or node2vec [8] are examples of shallow embedding algorithms. More complex embeddings can be obtained with Graph Neural Networks.

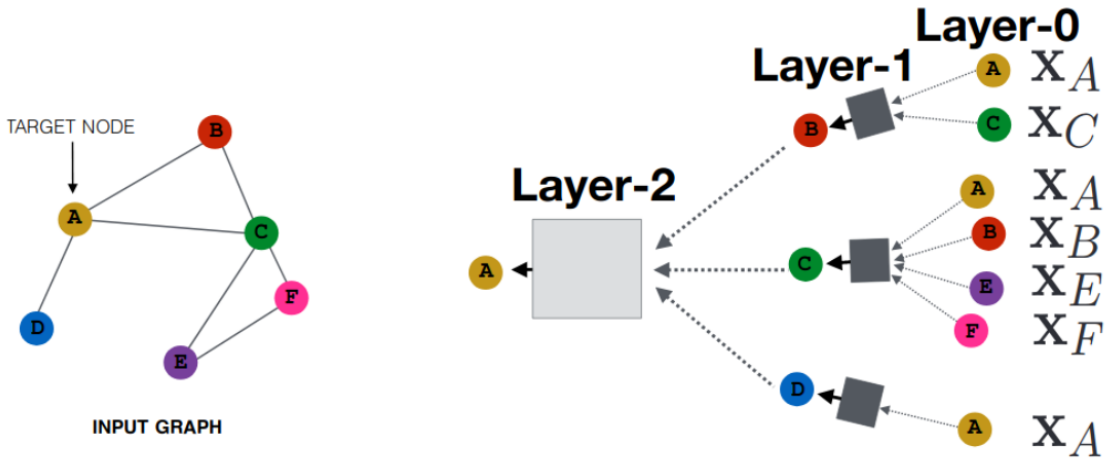

Figure 2.3: Layers of a Graph Neural Network, obtained from [2]. Left: A input graph, where A is the node to be explained. Right: Layers are created with the nodes at the same distance. Each new layer is based on the node of the preceding layer, starting from the node to be explained.

Graph Neural Networks (GNNs) [9] is a recent learning framework that brings deep representation learning [10] to graph and relational data. Another foundation for GNNs is the extension of a Convolutional Neural Network (CNNs) [11] to graphs.

The most relevant feature in GNNs is the use of neural message passing, in which messages are exchanged between the nodes and updated within the nodes with neural networks. The intuition is that we can generate node embeddings based on local network neighborhoods, so we have to aggregate information from the neighborhood.

Let's take GraphSAGE [2], a type of Graph Convolutional Network (GCN) with a slightly different architecture to allow trainable aggregation functions (not only standard convolutions). Every node, based on its neighborhood, defines a computation graph. The depth of the model is defined by the number of maximum hops we are considering for our neighborhood. Let K be the maximum number of hops, that means that we will have layers 0 to K. Layer K is located at the root node, layer K-1 is located in nodes which are 1 hop away from our root node and so on, until layer 0, which is located on the most distant nodes, as is depicted in Figure 2.3

In layer 0, the embedding is just the input feature of the node, and as we are traversing layers, the information is aggregated. A very basic approach can simply average neighbor messages and then apply the neural network:

$$\boldsymbol{h}_v^k = \sigma\left(\boldsymbol{W}_k \sum_{u \epsilon N(v)} \frac{\boldsymbol{h}_u^{k-1}}{|N(v)|} + \boldsymbol{B}_k \boldsymbol{h}_v^{k-1}\right), \forall k \epsilon \{1, ..., K\}$$

where $\boldsymbol{W}_k$ is the weight vector of the k-th layer and it is multiplying the average of neighbor's previous layer embeddings. $\boldsymbol{B}_k$ the bias vector of the k-th layer and it is multiplying $\boldsymbol{h}_v^{k-1}$, the previous layer embeddings. The nonlinear function is a sigmoid (σ).

We can see the parallelism with the basic structure of an artificial neuron as we have the weight term and the bias term, all enclosed in a nonlinear function.

The initial layer (0 layer) is created with the node features: $\boldsymbol{h}_v^0 = \boldsymbol{x}_v$. The embedding after the K layers is $\boldsymbol{h}_v^K$ and we denote it $\boldsymbol{z}_v$.

With this structure, we are ready to feed these embeddings in a loss function and run a stochastic gradient descent algorithm to train the parameters. There parameters are shared

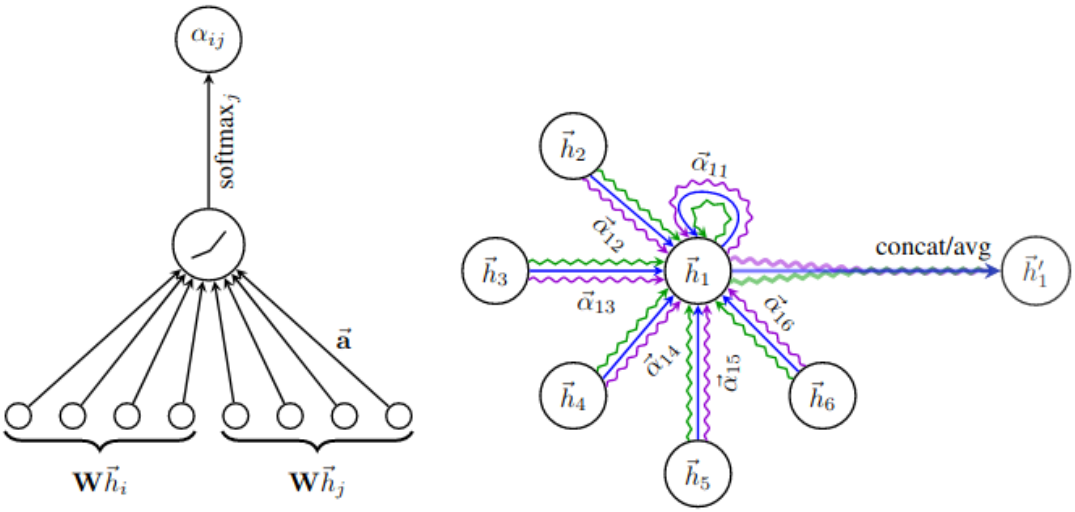

Figure 2.4: Graph Attention Networks, obtained from [12]. Left: The attention mechanism employed by GAT is parametrized by a weight vector and the non linear function is applied. Right: Conceptual flow of the multihead attention (3 heads). Each color represents different attention computations, which are then merged to obtain the new embedding.

for all the nodes and this allows to generalize for unseen nodes, allowing an inductive approach. Therefore, we can work with new nodes and generate their embeddings without retraining the network, which is really useful for many applications, especially those related to social networks.

Another approach is proposed in [12] under the name of Graph Attention Networks (GAT) and it is based on the concepts of Self Attention and Multi-Head Attention introduced by [13]. In GCN, the aggregation step consists in the normalized sum of neighbors' node features. In GATs the aggregation is not structure-dependent, but based on attention mechanism, as introduced in [14], that is, we can weight the importance of node′s features regarding to any other node in the computation graph, unleashing the structural constraints we have seen in GCNs, and achieving better generalization. In addition, multi-head attention allows for different computation of attention in the same node, which will be aggregated and averaged for a final result. This process is shown in Figure refGAT.

A drawback of these methods is that the neural network approach produces a black-box model which is difficult to interpret/explain. When we apply technology to social issues, it is unfair and unethical to apply measures ignoring the causes.

2.3 Explainability for Graph Machine Learning

In the context of Machine Learning, the terms explainability and interpretability are interchangeable in some studies [1] and differentiated in others [15]. As the definitions are not clear, we are not differentiating between the terms, so we will use them indistinctly. As posed by [16], interpretability is the degree to which a human can understand the cause of a decision. This cause can be interpreted as the answer to a why-question [17]. In terms of Machine Learning, this is viewed as a person asking why the prediction was taken and receiving an answer.

Following [16] ideas, good explanations are contrastive (as in [18]), selected, social, focus on the abnormal, truthful, consistent with prior beliefs of the explainee, general and probable. Each of these characteristics is not objective, therefore there is no mathematical definition of interpretability.

Based on the type of explanation we can differentiate two categories: instance-level meth-

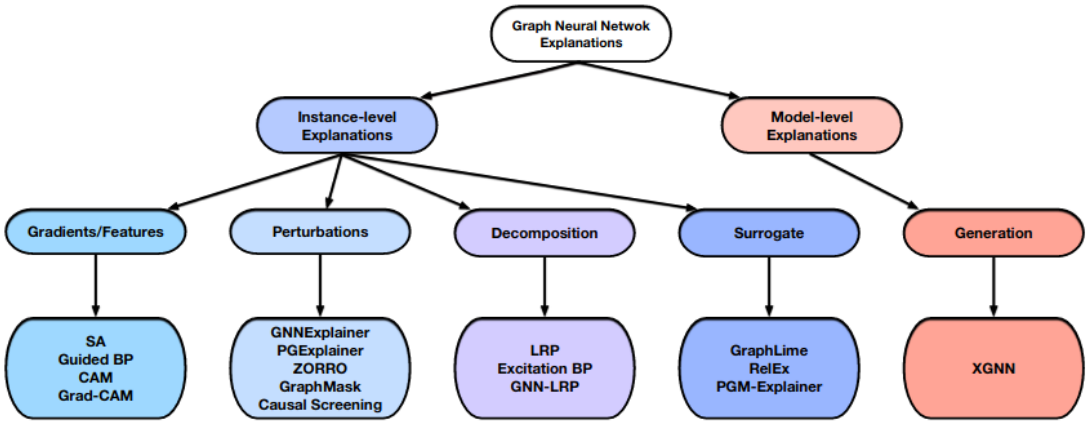

Figure 2.5: Taxonomy of explainers for GNNs, obtained from [19]

ods and model-level methods. The former explains one individual prediction while the latter explains the graph without respect to any specific input example. Regarding the methods for building the explainers for individual predictions, we can identify four groups according to [19], as shown in Figure 2.5:

- Based on Gradients
- Based on Perturbations
- Based on Decomposition
- Based on a Surrogate model

Focusing on the scope of this work, we can say that the concept we are using is post-hoc analysis of a prediction, where we are interested on knowing the relative weight of different nodes contributing to the prediction. We consider interpretability in the reduced scope of a useful debugging tool for machine learning models. We need to select metrics that acts as properties of our individual explanations, thus removing the need of a "human in the loop" to produce benchmarks and numerical comparisons.

There are a set of properties of a individual explanation proposed by [20]: Accuracy, Fidelity, Consistency, Stability, Comprehensibility, Certainty, Degree of Importance, Novelty and Representativeness. In the scope of this work we will study the stability, that represents how similar are the explanations for similar instances.

The methods compatible with a model-agnostic approach will be studied in the next section.

2.4 State of the art

The first approach in the literature is GNNExplainer [21], published on March 2019. It is therefore a very recent field and the publications are appearing at a faster pace. During the writing of this document, a survey was published at preprint status [19], and it will be used as a reference for completion.

The procedure for building the list is as follows:

(1) We analyze the references of the pre-published survey and we look for additional publications using the following query in Google Scholar:

- (explainer or explainability or interpretability or explanation) AND (gnn or "Graph Neural Network" or relational)

(2) There are 68 results as of December 2020. Within the results we look for publications where an explanation is obtained for an already existing GNN. We do not include in the study the methods or tools to improve explainability within the model or the application of the method to a specific field (i.e. biology, chemistry).

The filtered list is composed by ten publications:

- GNN explainer: A tool for post-hoc explanation of graph neural networks [21]
- XGNN: Towards Model-Level Explanations of Graph Neural Networks [22]
- Parameterized Explainer for Graph Neural Network [23]
- PGM-Explainer: Probabilistic Graphical Model Explanations for Graph Neural Networks [24]
- Contrastive Graph Neural Network Explanation (COGE) [25]
- GraphLIME: Local Interpretable Model Explanations for Graph Neural Networks [26]
- Evaluating Attribution for Graph Neural Networks [27]
- Interpreting and Understanding Graph Convolutional Neural Network using Gradient-based Attribution Method [28]
- Explain Graph Neural Networks to Understand Weighted Graph Features in Node Classification [29]
- XAI for Graphs: Explaining Graph Neural Network Predictions by Identifying Relevant Walks [30]
- RelEx: A Model-Agnostic Relational Model Explainer [31]

(3) Now we explore the article looking for references to prior works that are not in our list. These references are normally placed within the Introduction and Related Work sections.

- From [22] [24] and [25]: Explainability methods for graph convolutional neural networks [32]
- From [25]: Explainability Techniques for Graph Convolutional Networks [33]
- From [30]: Perturb More, Trap More: Understanding Behaviors of Graph Neural Networks [34]
- From [30]: Interpreting Graph Neural Networks for NLP With Differentiable Edge Masking [35]

These four articles were already inspected looking for new references, but all of the relevant references for our scope were already included in the list. Similarly, searching for new papers citing articles in the combined list produced no new findings.

From now on, we limit our scope to the methods based on perturbation or surrogate models that provide an open source implementation. We will analyze the 5 articles within the list that provide a public code repository.

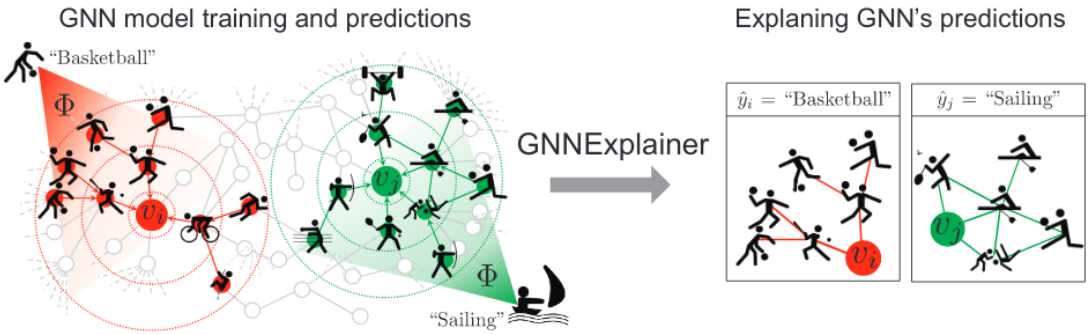

Figure 2.6: Conceptual idea of GNNExplainer, obtained from [21]

2.4.1 GNNExplainer

Name	GNN explainer: A tool for post-hoc explanation of graph neural networks		
First Author	Rex Ying	**Affiliation**	Stanford, USA
Date	2019 March	**Citations**	66

The method proposed by Rex Ying et al. is the first approach in explaining GNNs with the relational inductive bias and the features at the same time. It is based on Mutual Information (or Information Gain), a concept from information theory that the authors use to express how relevant is a set of features or edges in relation to a certain prediction. Since a *naive* approach (testing all possible sub-graph structures) is intractable, the authors use a Machine Learning approach to find the global maximum with the constraints of producing a compact and concise (small number of features and edges). The results are provided as masks for features and edges, that highlight the most relevant components, as shown in Figure 2.6.

There is an open source implementation from the authors and also an implementation in the Pytorch package. The latter is used for bench-marking in most of the articles related to GNN explainability, so we will use it. It is already shipped in version 1.5 and higher (May 25th 2020).

To use it, we have to instantiate the class "GNNExplainer" with the following parameters (as per documentation Pytorch 1.6.3 documentation):

- model (torch.nn.Module) – The GNN module to explain.
- epochs (int, optional) – The number of epochs to train. (default: 100)
- lr (float, optional) – The learning rate to apply. (default: 0.01)
- num_hops (int, optional) – The number of hops the model is aggregating information from. If set to None, will automatically try to detect this information based on the number of Message-Passing layers inside model. (default: None)
- log (bool, optional) – If set to False, will not log any learning progress. (default: True)

Optionally, we can set the coefficients to control the relative contribution of the different components of the loss function:

- edge_ent: Cross entropy objective between the label class and the model prediction regarding edges
- edge_reduction: Method used for aggregation of the number of edges (mean or summation)

- edge_size: Number of edges in the explanation
- node_feat_ent: Cross entropy objective between the label class and the model prediction regarding features
- node_feat_reduction: Method used for aggregation of the number of features (mean or summation)
- node_feat_size: Number of features in the explanation

To obtain the explanation, we should call the method explain_node which will return two masks, the first for the edges and the second for the features. To help with the visualization, this implementation comes with a visualize_sub-graph method which plots the computation graphs highlighting the relevant edges.

2.4.2 PG-Explainer

Name	Parameterized Explainer for Graph Neural Network		
First Author	Dongsheng Luo	**Affiliation**	Pennsylvania State University, USA
Date	2020 November	**Citations**	0

In this method the authors focus only on the relational structure (features can be explained with non-graph explainability methods) and proposes a generative probabilistic mode, to learn the underlying structures from the observed graph data (the latent variables in edge distributions). It can be used for explaining individual instances, groups of instances and the whole graph.

The main advantages cited by the authors are:

- It does not need to retrain for generating explanations for different nodes, so it is a faster method
- It can collectively explain multiple instances

To learn it, the objective function is similar to GNNExplainer as it is using the Mutual Information. It considers that the explanation graph is a Gilbert Random Graph (selection of an edge is independent from other edges) and instead of using binary values for the edges, it uses continuous values in the range [0,1]. To learn, it used the reparametrization trick to employ gradient-based methods, and then a Monte Carlo approach is used to approximate the objective function. To control the smoothness of the values of the edges, the parameter τ (for temperature) is used ($\tau = 0$ means binary values, and as it increases the values are smoothed).

The key idea is using a graph generative model to create the explanations. The generation is based in the latent structure which is learned from the whole graph, and this is why it does not need to retrain on every node. The process is shown in Figure 2.7.

This implementation is available at Github[1]. It uses Python and comes with all the relevant notebooks for the examples shown in the article. It uses Keras (on top of Tensorflow 2.0). It is presented as a "Explainer" class than can be instantiated with the following parameters (obtained directly from the source code):

- coff_size: Coefficient for size constraint
- coff_lap: Coefficient for Laplacian
- coff_ent: Coefficient for entropy loss

[1]PG-Explainer: https://github.com/flyingdoog/PGExplainer

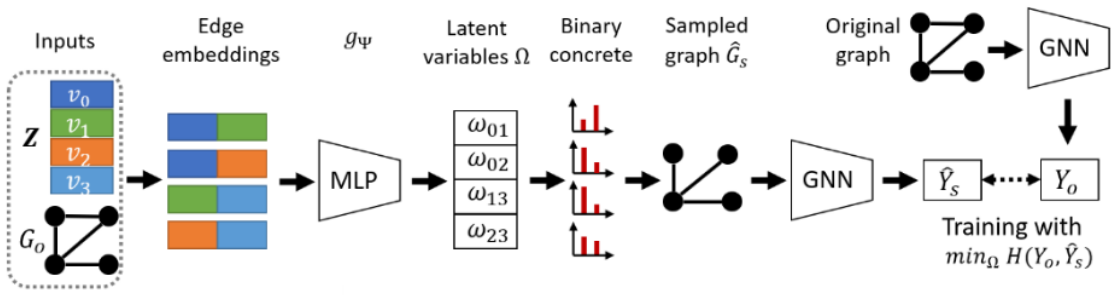

Figure 2.7: Flow of PG-Explainer, obtained from [23]. From left to right: Inputs are the original graph G_o and the embeddings Z, which are used to compute the latent variables in edge distributions (Ω), which are then binarized. We then select the top edges according to Ω. We sample a random graph $\hat{G}_s$ which is fed into the GNN to get the prediction label and optimize the parameter Ψ in our model using the cross-entropy between the original prediction Y_o and the updated prediction $\hat{Y}_s$.

- weight_decay: Weight for L2 loss on embedding matrix
- miGroudTruth (true/false): Mutual Information between hat y and Ground Truth Label
- coff_t0: Initial temperature
- coff_te: Final temperature
- sample_bias: Bias for sampling for 0-1

It provides a method "call" that returns the mask of the nodes relevant for the explanation as a list of nodes.

The datasets used for evaluations are those of GNNExplainer.

2.4.3 COGE

Name	Contrastive Graph Neural Network Explanation		
First Author	Lukas Faber	**Affiliation**	ETH Zurich, Switzerland
Date	2020 October	**Citations**	0

This article leverages the concept of Distribution Compliant Explanation (DCE), which consists on only ever use data consistent with the training distribution for making model explanations. Therefore, instead of perturbing existing graphs, it bases its explanation on other graphs from the training dataset.

The key idea is finding the parts of the graph that makes it distant to graphs with a different label. To measure this distance, the Optimal Transport (OT) distance is employed, because it is able to compare embeddings of two graphs on node granularity. The optimization consists on jointly maximizing OT distance to graphs with the same label and minimizing OT distance to graphs with a different label.

The implementation is available at Github[2].

For node explanation, there is the class *ExplainMethod* which allows four different explanation methods. We focus on the contrastive explanation, which calls the method *constrast* with the following parameters:

[2]COGE: https://github.com/lukasjf/contrastive-gnn-explanation

- loss_str: add each of '-', '+' and 's' for different parts of loss
- similar_size: number of similar graphs to use for positive and negative set
- distance_str learning: distance measure to use can be one of ['ot,'avg'] (OT is for Optimal Transport)

The explain method returns the adjacency mask highlighting the relevant nodes.

The datasets used for evaluations are those of GNNExplainer, and the implementation of GNNExplainer is not the Pytorch's one, but the original implementation[3] from the article authors.

2.4.4 PGM-Explainer

Name	PGM-Explainer: Probabilistic Graphical Model Explanations for GNN		
First Author	Minh N. Vu	**Affiliation**	University of Florida, USA
Date	2020 October	**Citations**	0

The main contribution of the paper is approximating the target prediction with a graphical model. A Probabilistic Graphical Model (PGM) is a statistical model for which a graph expresses the conditional dependence structure between random variables. In this paper the selected PGM is a Bayesian Network, which employs a directed acyclic graph (DAG). The procedure is conducted in three steps:

- Data generation
- Variables selection
- Structure learning

In the data generation phase, some of the input data is perturbed with the mean value among all nodes. The variable selection consists on looking for the optimal Bayesian Network, but this problem is intractable for most of the networks due to its search-space size. To handle this limitation, we just need to find a subset which contains the minimum Markov-blanket and is concise enough, which is conducted using pairwise independence tests. To learn the structure, we use the Bayesian Information Criterion (BIC) as an objective function, along with hill-climbing algorithm.

PGM-Explainer can be used both for node and graph explanation. Its architecture is depicted in Figure 2.8.

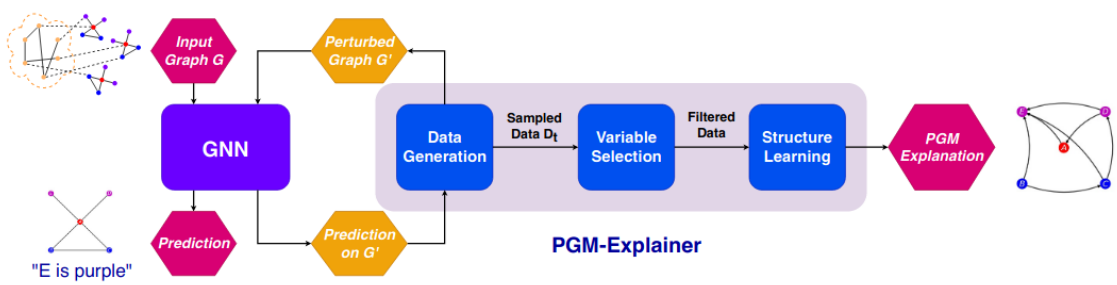

Figure 2.8: Architecture of PGM-Explainer, obtained from [24]

[3]GNNExplainer: https://github.com/RexYing/gnn-model-explainer

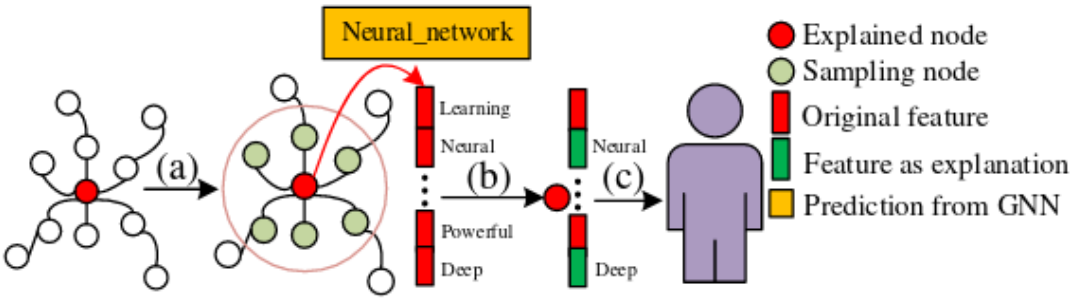

Figure 2.9: Conceptual flow of GraphLIME, obtained from [26]

The implementation is available at Github[4].

For node explanation, there is the class Node_Explainer and the method "explain" that returns the nodes and the conditional probabilities associated with each of them.

The datasets used for evaluations are those of GNNExplainer.

2.4.5 GraphLIME

Name	GraphLIME: Local Interpretable Model Explanations for Graph Neural Networks		
First Author	Qiang Huang	**Affiliation**	Jilin University, China
Date	2020 January	**Citations**	13

The mechanism in GraphLIME consists in three steps:

1. Predict the classification for all the nodes in the neighborhood, then generate a tabular dataset with the features and the classification for each no
2. Use the tabular data as input for HSIC Lasso, a feature selection method
3. Select the top K features from the HSIC Lasso

HSIC Lasso is well suited for high-dimensional feature selection, meaning that the number of features should be much higher than the number of samples. Particularly, as stated in [HSIC Lasso], for this method to be efficient, the number of features should be at least the squared number of samples (this happens only for a fraction of the nodes in the analyzed datasets).

This method is limited to the features of the nodes, the only relational input is filtering the nodes outside the neighborhood this there is no subgraph output as in GNNExplainer. A conceptual flow is depicted in Figure 2.9.

The implementation is available at Github[5]. The language of the implementation is Python, it is using Pytorch and the dataset used for testing is Cora. It is presented as a "GraphLIME" class than can be instantiated with the following parameters (obtained directly from the source code):

- hop: Maximum distance between the explained node and the neighboorhod

[4]PGM-Explainer: https://github.com/vunhatminh/PGMExplainer
[5]GraphLIME: https://github.com/WilliamCCHuang/GraphLIME

- rho: Constant that multiplies the penalty term.

It provides a method "explain_node" that returns a list of the coefficients for every feature (a higher coefficient means more relevance).

2.5 Computer methods

The mathematical foundations for this work are aligned with the courses of the Master of Sciences where it belongs, but a few of the analyzed tools employ methods from another disciplines. For the comparison of different explanations we need to gather tools from Information Theory, in particular Mutual Information, Entropy and Cross-entropy. We have found also interesting the current work in line with the Bayesian Networks and the Probabilistic Graphical Models, which accounts for the uncertainty of the explanation process.

Most of the computer methods found in the state of the art are using Python for its implementation. Python is a programming language very popular in Data Science. It is interpreted, object-oriented and a high-level programming language with dynamic semantics. The community is very active and there are active developing in many areas of Machine Learning (ML). In particular, Pytorch and Keras are two popular frameworks for ML that are built on Python. The main features of Pytorch are tensor computing with GPU acceleration and Deep neural networks with automatic differentiation system. Keras is a Machine Learning API, running on top of the machine learning platform TensorFlow, which is a symbolic math library based on data-flow and differentiable programming.

It is shown that the differentiable programming paradigm is very valuable in Machine Learning [36] (the use of gradients are a good example of the necessity), so we should also name recent packages in the Julia programming language like Zygote. Although we are not using Julia in this work, it is a language perfectly suited for Machine Learning. The reason we are not using it is because we are based on existing implementations of explainers for Graph Neural Networks written in Python.

Most of the studied tools use the ADAM optimizer [37] (an algorithm for first-order gradient-based optimization of stochastic objective functions, which yields very good results while it is computationally efficient and has little memory requirements). We use the interface provided with Pytorch so the reader does not need to know about the internals, just the typical pipeline for the training process (feed forward, gradient and backward steps).

During the development we use the Monte Carlo method [38], which is a computational algorithm based on repeated random sampling to obtain, in this case, a probability distribution. The idea is reducing the instability by aggregating the values in the obtained distribution.

2.6 Datasets

In the next sections we are going to use two datasets that represent citation networks: Cora and Pubmed.

The Cora dataset from [39] contains 2708 nodes representing scientific publications. Each node has a label which represents its class (a number between 1 and 7). There are 5429 edges in the graph. The features of each node are binary, indicating the presence(1) or absence(0) of a 1433-words dictionary.

The Pubmed dataset from [40] contains 19717 nodes representing scientific publications from PubMed database pertaining to diabetes classified into one of three classes ("Experimental", "Type 1", "Type 2"). There are 44338 edges. The features of each node are the term

frequency–inverse document frequency (tf-idf [41], a numerical statistic related to how relevant is a word to a document in a collection) on a 500-words dictionary.

Chapter 3

Explainability of GNNs

In this chapter we present the problem of explaining a prediction conducted by a GNN model. We select a tool and find that for a certain range of problems it is not working as expected.

3.1 Problem statement

Given the following objects:

1. A graph G = (V,E) where V is an n-element set of vertices, and for any two pairs of vertices u, v ∈ V there is an edge e ∈ E whose ends are u and v. Additionally, F is a n-element set of features, so each set is assigned to a vertex. F can be understood as a matrix where each row represent the features of a node.
2. A Graph Neural Network Model ϕ trained with the graph G and matrix F, with the labels L associated to the nodes
3. A set of labels L (one per node)

The prediction of the GNN is the label of the node. The explanation of the prediction on a certain node gives information about the most relevant elements of V,E,F that the model ϕ is using for its prediction.

In particular we are studying the explainability methods with the following properties:

- Post-hoc: It conducts the analysis after the inference.
- Model-agnostic: It can be applied to any Graph Neural Network already trained.
- Instance-wise: It is explaining a single prediction, not the whole graph.
- Classification problem: The result is only one type within a fixed set of possible types.
- Minimal Sufficient Subset: The explanation is a subset of the graph (a set of vertices) and a subset of features, which are the most relevant for the inference
- Perturbation-based approach: It employs the occlusion method, where some entity is deactivated to check if it is relevant or not. If the accuracy is down after some occlusion, that that entity is considered relevant to the degree of the accuracy decrease.

When we talk about an explanation for a classification problem in Graph Machine Learning in terms of the attribution problem (finding the more relevant parts of an input that are responsible for the prediction) we have four different types of entities:

- Features
- Links (or edges)
- Nodes (or vertices)
- Labels

These entities are part of the computation graph around a node. A computation graph is the induced sub-graph containing only the nodes that can be reached from a root node within k hops and the edges between them. It can also be seen as an egocentric network of degree k. We denote it as G_s

Our algorithms will not use the complete graph, only the computation graph, ignoring the rest of the graph. This local interpretation allows managing huge dataset with reduced computation needs and it comes from the idea that the most relevant influence for a node comes from its vicinity.

When a GNN working in a node classification task predicts a certain class, we are interested in the combination of edges that the GNN is taking into account for this decision. If we consider that an edge has a certain grade of relevance, the attribution consists on extracting this notion of relevance as a number for each edge.

The different combinations of the relevance forms a search space. Finding the best attribution combination consists on iterating through the search space, evaluating each combination with an objective function and selecting the one with the optimal value, according to some goals and constraints (i.e. the number of elements).

3.2 Need for heuristics

3.2.1 Determining combinations within the computation graph

For an exhaustive search of all possible attribution combinations, we will start considering only the nodes (without taking into account the edges or the features). Let S be a set containing the nodes of the computation graph G_s, where |S| denotes the cardinality of S ($|S| = n$). Then:

$$|P(S)| = 2^n$$

where P(S) denotes the power set of S. The power set of a set, denoted by P(S) is the set of all possible subsets of S, including the empty and the set itself, hence all possible attribution combinations.

We can also understand the attribution problem as building a mask for every type of input part. The mask will be a set of real values, one pear each node or edge, that will define the relevance on the entity. We can understand the mask as a filter that overlapped with the original set, will only pass the relevant entities.

We can simplify for now, and use a binary mask where a '1' indicates that the instance is relevant while a '0' indicates its lack of importance. For the nodes, the mask will have a size n (the cardinality of the set, which is the number of nodes) and therefore the number of different combinations of the mask is 2^n.

For a GNN, which is based on message passing through the edges, this is not a valid input; we have to take into account the edges. If we follow the same strategy of building a mask for the edges, we will have a size of search space equal to 2^e where e is the number of edges.

In Graph Theory, the Degree centrality is defined as the number of links incident upon a node [42], while the beta index of a Graph [43] measures the level of connectivity in a graph and

is expressed by the relationship between the number of links over the number of nodes. In some of our areas of interest, as social networks, beta will be higher than 1, therefore the number of edges in the computation graph will be much higher than the number of nodes. The average number of connections of a Facebook user was 338 in 2014 according to [44].

Depending upon our budget in computational resources and time, there is a upper bound of the number of edges that we can analyze with the exhaustive approach.

$$executionTime * 2^e <= budgetTime$$

$$e = log_2(budgetTime/executionTime)$$

On a personal computer with a GPU, a typical execution time of the inference model using a GNN of the type Graph Convolution Network (GCN) trained with Cora dataset is 0.005 seconds. For debugging, we can set a reasonable time for waiting as 10 seconds. With these values, we obtain a maximum number of edges of 11 (10.966). In the Cora dataset, with k set to 2 (the computation graph contains all nodes within 2 hops, and the edges between them) and using only the giant component (2485 out of 2707 nodes), the number of nodes with a degree (within its computation graph) of 11 or less is 552. Even if we increase the waiting time to 60 seconds, we obtain 13 edges (13.55), which covers only a quarter of the nodes in the principal component (640). This is due to the logarithm in the equation, as we increase the budget n times, we only obtain $log_2(n)$ edges in addition:

$$edges = log_2(n * \frac{budgetTime}{executionTime}) = log_2(n) + log_2\frac{budgetTime}{executionTime}$$

Therefore we need an approach different than brute-forcing, even if we can use a computer multiple times faster. For a simple dataset as Cora, the node with the highest number of edges in its computation graph for a cutoff of 2 is 895, giving us a search-size size of $2,641472656 \times 10^{269}$.

3.2.2 Reducing the search space

A first heuristic can be reducing the search-space size by applying constraints. As already mentioned, GNNs makes use of the message passing mechanism, so we can consider only combinations that form a single connected component where at least one of the edges is connected the root node (the node that is being explained). The intuition is that an edge is activated if it is passing a message that is relevant to the prediction. If there is no path connecting to the explained node, there is no possible impact on the prediction.

To study this constraint, which is highly dependent on the graph structure, we are going to start with the best-case scenario. The minimum number of edges in a connected graph (the computation graph is connected by definition) is n-1. If we consider that our computation graph is a tree, where the root is the explained node, it will have n-1 edges, thus the minimum number of edges. Any other structure will have equal or higher number of edges.

Let T be a perfect binary rooted tree. Now consider we can remove any number of edges, and that by removing an edge we also remove the part of the graph that is not reachable from the root node. The number of different combinations where all the edges pertain to only one connected component where at least one edge is connected the root node is given by the following recursive equation:

$$B(T_l i) = \prod(1 + B(T_l i))$$

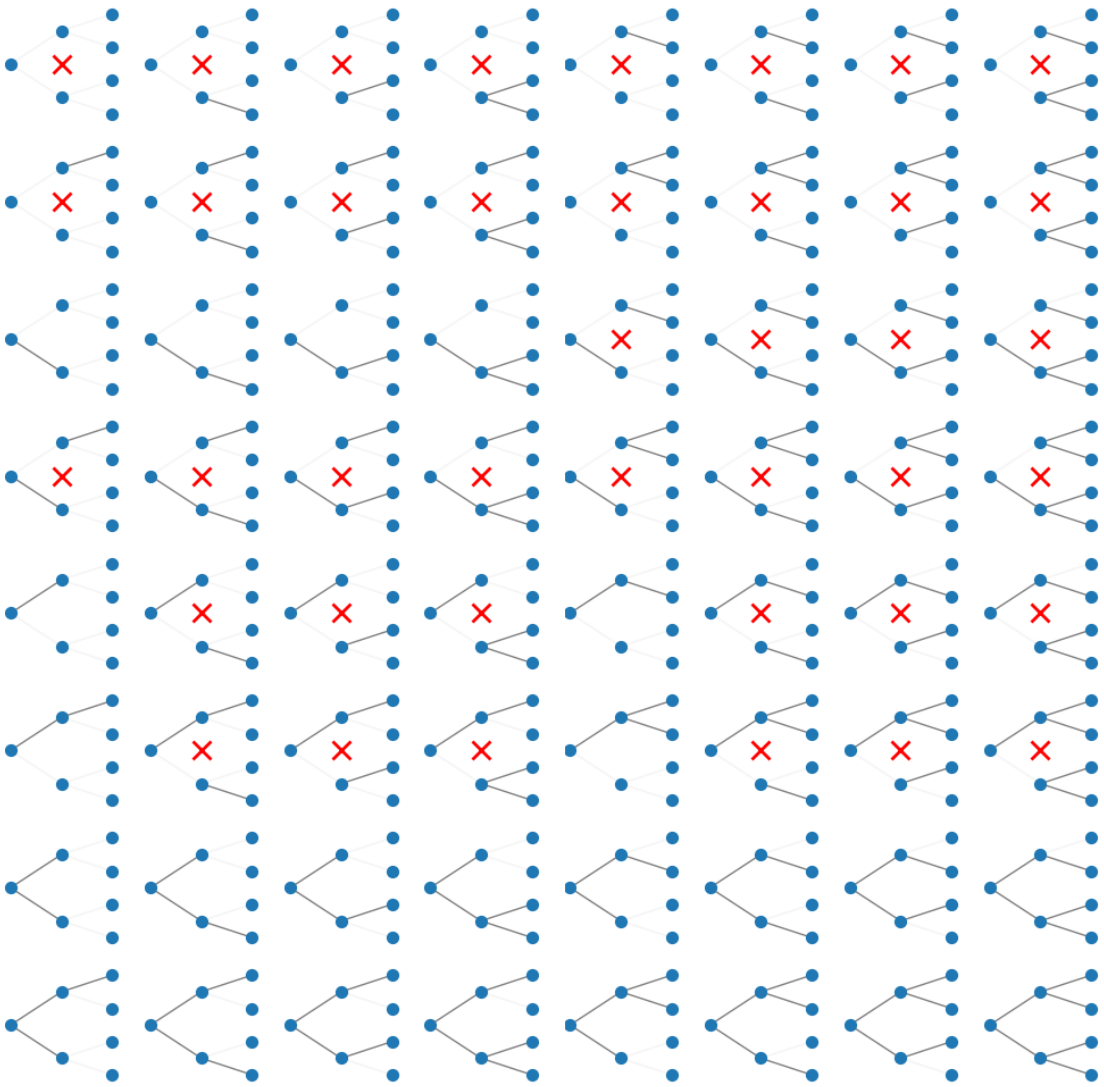

Figure 3.1: Search space for a rooted binary tree of two levels. Combinations not allowed are marked with a cross because there are two or more connected components and/or do not connect the root node

For example, the rooted tree in Figure 3.1, out of 64 possible combinations, only 24 are allowed. For a tree with 15 nodes and 14 edges, where the power set is composed by 16384 combinations, the allowed combinations are just 625, that can be evaluated in about 30 seconds in the aforementioned scenario. Even with the best-case scenario, this improvement is not enough for most of the nodes of real-world datasets, and we need to set more constraints.

Following the same strategy as in the first constraint, we know before-hand that the size of the explanation (the number of entities of a certain type) must be a reasonable number for a person. While this is not quantifiable and it can vary from person to person, we can follow Miller's Law conception about short-term memory and set a number between 5 and 9. Also, if we have any information on the domain, we can use that information to look for explanations of a certain size.

With these constraints we are increasing the computational charge into finding the appropriate combinations in order to reduce the computation effort in the evaluation stage. We must take into account this trade-off to calculate the maximum size of the computation graph that we should evaluate with the exhaustive approach. Moreover, it can be useful for comparison with other methods, acting as a ground truth for the optimization problem.

3.2.3 Quantifying the need of heuristics in citation network datasets

To assess to what extent we can cover the datasets with the different approaches, we can compute the size of the computation graphs of every node in the Cora and Pubmed datasets (we are only considering the giant component of every dataset).

The size of the giant componet are the following:

- Cora giant component: 2485 nodes
- Pubmed giant component: 19717 nodes

Depending on the degree of the root node, we can have three different scenarios:

Case 1: For computation graphs with a small number of edges (i.e. less than 9) the constraint of the number of edges is not applicable. Returning every edge as an explanation is misleading, because the explainer is not really necessary. The number of nodes in the each network taht fall in this case is the following:

- Cora: 387 nodes (15.6%)
- Pubmed: 1647 nodes (8.3%)

Case 2: For computation graphs with a number of edges between 9 and 14 (this number depends on the computation power), we can use the exhaustive approach.

- Cora: 304 nodes (12.2%)
- Pubmed: 2479 nodes (12.6%)

Case 3: For computation graphs with higher number of edges (15 or higher) we need a heuristic technique.

- Cora: 1794 nodes (72.2%)
- Pubmed: 15591 nodes (79.1%)

Therefore, given the large number of nodes that fall in scenario 3, it is clear that the exhaustive approach will not able to deliver timely results for these two real-word datasets, and that we need to find another approach.

3.3 Need for stability

We select the GNNExplainer method for being the baseline of every other method and because it is widely used as a Pytorch method. GNNExplainer is using a heuristic technique for arriving at a compact explanation. The steps are the following:

- For every node to be explained it obtains its computation graph and the initial prediction
- It jointly learns the mask for edges and the mask for features with an Adam optimizer and a loss function based on 5 components (prediction, edge entropy, number of edges, feature entropy and number of features).
- Returns the edge mask and feature mask

This method allows obtaining approximate results for large computation graphs within a reasonable execution time. There is an implementation of the method included in the Pytorch Geometric package. In this work we will focus on this implementation.

A starting example is provided with the Cora dataset. The example sets a simple architecture with two layers of Graph Convolutional Operators (GCNConv), a Rectifier Linear Unit, a dropout and returns the values after applying logsoftmax. With this architecture it trains a model for node classification with the Adam optimizer on the training split of the Cora Dataset. Once it has the model, it creates a GNNExplainer object and calls its method "explain_node". The output of the method comprises the masks for features and edges, which can be visualized with the "visualize_subgraph" method.

If we execute the example script, which is set to explain node number 10, two consecutive times, we obtain the plots of Figure 3.2. The color of the nodes represent the label of the node and the color of the edge represent its relevance, being grey irrelevant and black relevant. We can quickly spot that the layout has changed, that is, the nodes are in different locations. Moreover, if we compare the edges between the two images, we see that there a some differences. Take for example the edges connecting nodes 306 and 476. These edges are considered relevant in the first explanation, but not at the second one.

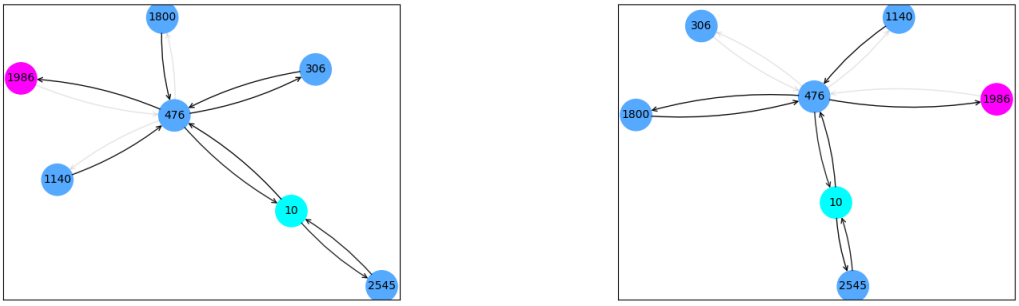

Figure 3.2: Two consecutive executions for explaining node 10 at Cora Dataset. In the first execution (left picture) edges from and to node 306 are relevant (black color), while in the second execution (right picture) they are irrelevant (grey color). Node color represent the label of the node.

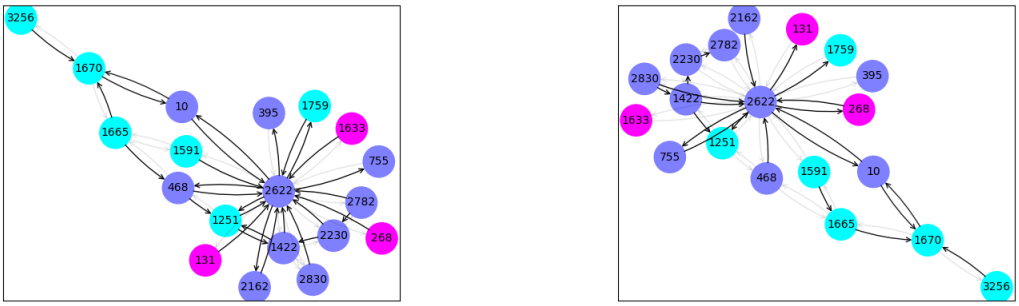

Figure 3.3: Two consecutive executions for explaining node 10 at Pubmed Dataset. In the first execution (left picture) there are 31 relevant edges (black color), while in the second execution (right picture) they are 22 relevant edges (grey color). Node color represent the label of the node.

We can repeat the experiment using another dataset, like Pubmed, and obtain the plots at Figure 3.3. In this case, we can see not only that there are different edges in both plots, but also that the number of edges is too high to understand the explanation.

Theses differences are the result of a lack of stability and conciseness (both desired properties for explainers). In the next chapter we will analyze the origin of the issues and will propose a method to improve it.

Chapter 4

Understanding and improving explainability of GNNExplainer

As shown in the previous chapter, the GNNExplainer implementation of the Pytorch project lacks stability in a certain range of nodes within the citation network datasets. In this chapter we analyze the origin of the instability and propose an improvement. Finally, we show the effect of our modifications for different configurations.

4.1 Effect of randomness in GNNExplainer

The non-deterministic behaviour shown in the previous section is probably the result of stochastic functions (i.e. pseudorandom generators) within the code. If we look carefully we find that the initialization of the masks for the edges and the features is done with the torch.randn method, which returns a tensor filled with random numbers from a normal distribution with mean 0 and variance 1.

$$out_i \sim \mathcal{N}(0, 1)$$

By multiplying the output we can set the variance. As we can see in the code snippet below, the variance is fixed to 0.1 for the features and $sqrt(2.0) * sqrt(2.0/(2 * N))$ for the edges. The first term comes from $torch.nn.init.calculate\_gain('relu')$ and it is used to scale the standard deviation with respect to the applied non-linearity (in this case a Rectifier Linear Unit [45]). Generally, the spread of the random distribution will be higher with a small number of edges and will shrink as this number grows. The number of elements of the edge mask is the number of edges, and the number of elements of the feature mask is the number of features.

Listing 4.1: Code for mask setting in GNNExplainer

```
def __set_masks__(self, x, edge_index, init="normal"):
    (N, F), E = x.size(), edge_index.size(1)

    std = 0.1
    self.node_feat_mask = torch.nn.Parameter(torch.randn(F) * 0.1)

    std = torch.nn.init.calculate_gain('relu') * sqrt(2.0 / (2 * N))
    self.edge_mask = torch.nn.Parameter(torch.randn(E) * std)

    for module in self.model.modules():
        if isinstance(module, MessagePassing):
```

```
module.__explain__ = True
module.__edge_mask__ = self.edge_mask
```

Other sources for randomness may come from the training phase. To explore how accountable is each section of code, we insert a manual control of the random seed at different lines of code. Also, we set the property cudnn.deterministic to True to only use deterministic implementations.

The manual control of the random seed consists on inserting the method "torch.manual_seed" at the beginning of the script, with a fixed value i. Then we surround the masking setting function with a manual seed j at the beginning and i at the end, meaning that only this specific function will use the seed j, while the rest of the code will use seed i.

We proceed testing (running multiple times) and comparing the real values of the edge mask with different strategies:

1. Same seed for all the code: No difference in outcome
2. Seed in masking setting function remains unchanged, seed for the rest of the code takes different values each time: Difference in the outcome less than 1%
3. Seed in masking setting function takes different values each time, while seed for the rest of the code remains unchanged: Difference in the outcome less up to 90%
4. Both seeds take different values each time: Difference in the outcome less up to 90%

Therefore it seems that the randomness shown in the example comes exclusively from the the function "__set_masks__".

Next, to try quantifying the instability and discover its origin, we are going to introduce probes within the process, without modifying it. We recreate the GNNExplainer code in our local directory and set our code to import this file instead of the original package from Pytorch. We modify the "explain_node" function to introduce the edge_mask_tracker list, which is going to store the value of the edge mask at each epoch of the training. With this list, we can plot the evolution of the edge mask value. We set to plot multiple executions (one hundred) to account for the randomness in the initialization.

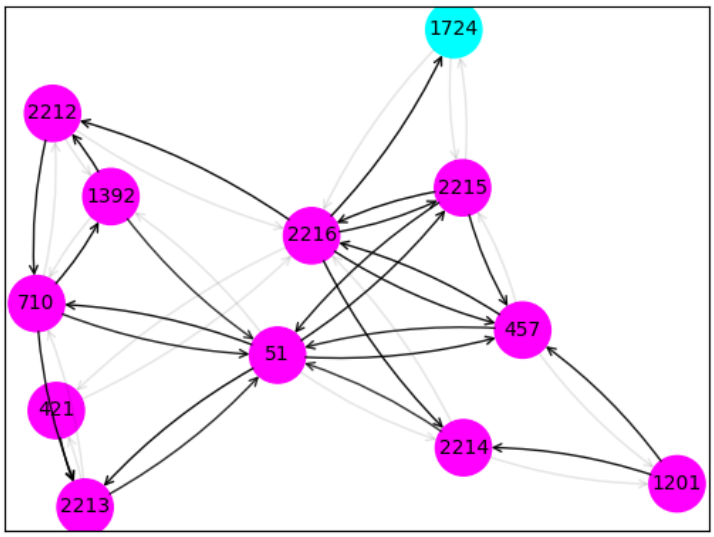

Figure 4.1: Explanation for node 51 at Cora dataset, with label 1 as pink and label 2 as blue

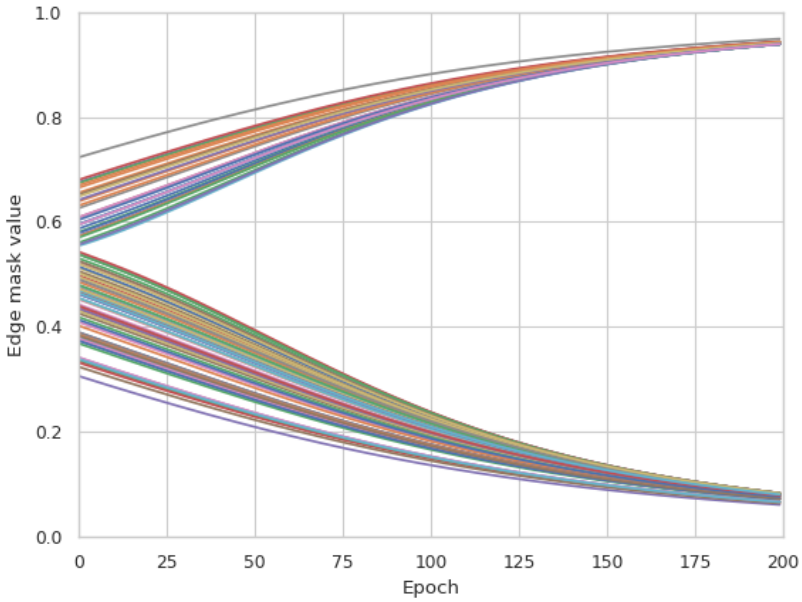

Figure 4.2: Evolution of the edge mask real value of a single edge during multiple consecutive runs. The edge is connecting node 421 with node 2216, while explaining node 51 of Cora dataset.

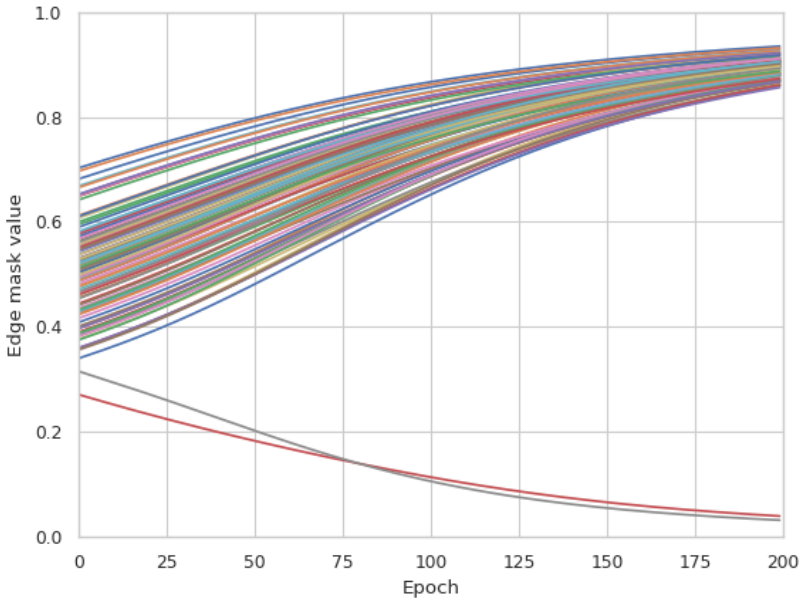

Figure 4.3: Evolution of the edge mask real value of a single edge during multiple consecutive runs. The edge is connecting node 51 with node 710, while explaining node 51 of Cora dataset.

We can see in Figure 4.2 that the spread of the initial values depends on the number of edges within the computation graph (N). In this case $N = 12$ so $std = sqrt(2.0) * sqrt(2.0/(2 * N)) = 0,408$. It is also clear that there is a cutoff value from where the edge mask values will go to higher values or lower values. This can be seen as a bistable behaviour, and this bistability (near one and near zero) produces the instability across multiple executions. The cutoff value is placed in different locations for different edges at the same node.

We explain node 51 at Cora dataset, as shown in Figure 4.1. The analysis of edge connecting node 51 with node 710 (Figure 4.3) shows a much higher stability than the edge connecting node 421 with node 2216 (Figure 4.2).

To know how frequent is this behaviour within our datasets, we define the explanation of an edge as inconsistent if it exhibits the bistable behavior at least one time out of ten runs.

We are interested on the proportion of edges within the explanation of a node that provide inconsistent explanations with regards to N. We set the experiment and produce the plot at Figure 4.4.

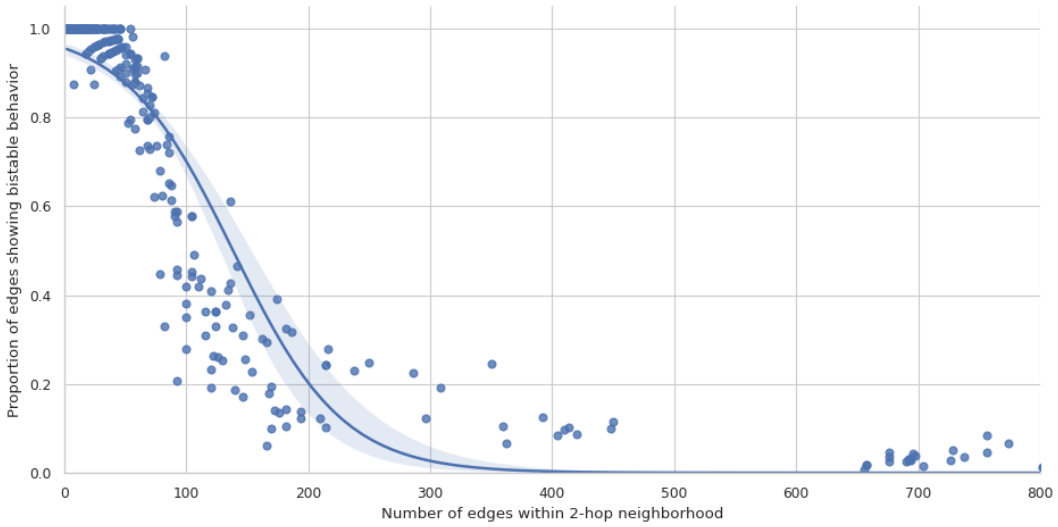

Figure 4.4: Proportion of bistable behaviour in edges regarding N (Cora dataset, 300 uniformly sampled nodes). Linear scale

We have used a logistic regression as it fits properly the distribution. It is clear that N is correlated with the proportion of the edges showing an inconsistent explanation. There are two parts of GNNExplainer directly linked with this size:

1. The spread of the Gaussian variable that populates the edge mask depends on N. As N is smaller the spread is larger, leading to more probable bistable behaviour as the cutoff value has a bigger range to lie in.
2. One of the components of the loss function depends on N, and it is multiplied by the same coefficient regardless N value. This leads to little effect on the overall loss function when N is small.

As we intend to leave the main part of the code for GNNExplainer untouched for our improvement, we leave the first dependency out. For the second dependency, we can modify the coefficients from outside the class, so we will explore in the next section how to improve the stability.

4.2 Effect of the number of edges in the loss function

The loss function is the objective function to minimize, therefore it will shape how the algorithm modifies the parameters to find a local minima which is acceptable (ideally a global minima). In GNNExplainer, the loss function obtained from the combination of 5 different functions or values:

- Prediction loss function: Log-logits of the model's output confidence for the predicted label
- Value related to the number of edges
- Value related to the number of features
- Value related to the entropy of the edges
- Value related to the entropy of the features

The first value (prediction loss function) is used for the loss function without any parameter, but the last four values are the result of multiplying the referenced value (edges or features) by a fixed coefficient. Of these four values, two of them also use reduction functions that aggregate the values using the summation or the mean.

Listing 4.2: Coefficients in GNNExplainer

```
coeffs = {
    'edge_size': 0.005,
    'edge_reduction': 'sum',
    'node_feat_size': 1.0,
    'node_feat_reduction': 'mean',
    'edge_ent': 1.0,
    'node_feat_ent': 0.1,
}
```

In this work we focus on the edge_size coefficient, as it is the one affecting the behaviour under study. We modify the loss function to return each value separately, then log the values for each epoch to plot the evolution. Note that this modification only produces new lists, but it does not affect at the loss itself.

Listing 4.3: Code for our modified loss function in GNNExplainer

```
def __loss__(self, node_idx, log_logits, pred_label):
    pred_loss = -log_logits[node_idx, pred_label[node_idx]]

    m = self.edge_mask.sigmoid()
    edge_reduce = getattr(torch, self.coeffs['edge_reduction'])
    edge_loss = self.coeffs['edge_size'] * edge_reduce(m)
    ent = -m * torch.log(m + EPS) - (1 - m) * torch.log(1 - m + EPS)
    edge_ent_loss = self.coeffs['edge_ent'] * ent.mean()

    f = self.node_feat_mask.sigmoid()
    node_feat_reduce = getattr(torch, self.coeffs['node_feat_reduction'])
    feat_loss = self.coeffs['node_feat_size'] * node_feat_reduce(f)
    ent = -f * torch.log(f + EPS) - (1 - f) * torch.log(1 - f + EPS)
    feat_ent_loss = self.coeffs['node_feat_ent'] * ent.mean()

    loss = pred_loss + edge_loss + edge_ent_loss + feat_loss + feat_ent_loss
    return loss, pred_loss, edge_loss, edge_ent_loss, feat_loss, feat_ent_loss
```

We execute the example script for explaining individual node 10 in Cora dataset. We track the values of the loss and its components and log them into the "loss_evolution.csv", which is then plotted. The result is shown in Figure 4.5. We can notice that the effect of the component linked with the edge size (the number of edges) is irrelevant from the beginning. This is not what we can expect from the component which is related to the size of the explanation.

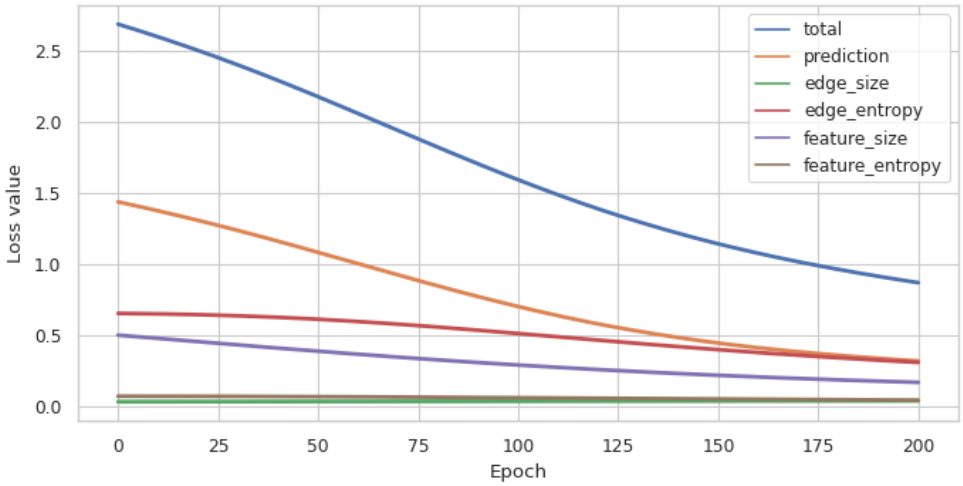

Figure 4.5: Evolution of the contribution to the loss function of different values for node 10 at Cora with default settings

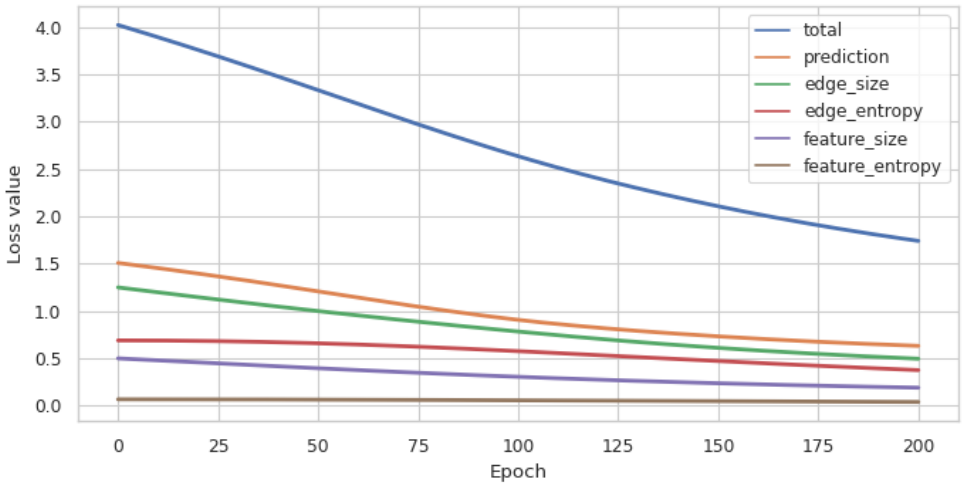

Figure 4.6: Evolution of the contribution to the loss function of different values for node 112 at Cora with default settings

The reason of this absence of effect is the small value of the number of edges within the 2-hop neighbourhood of the node (from now on we will refer to this as N) combined with the small value of the coefficient "edge_size". As the reduction function is a summation, the coefficient should be small enough to make the contribution to the total loss in accordance to the rest of components. If it is too high, it will make other components irrelevant; if it is too small, it becomes irrelevant itself. In this case we face the issue of a fixed coefficient of 0.005, which multiplied by N is just 0.06 for the maximum scenario (all edges with a value of 1.0). In comparison with other components this is just too small to be taken into account by the optimizer.

We are now selecting a counterexample, a node where N is large. We select node 112 with N=500. At the beginning, as we are populating the edge mask with values obtained from a Gaussian

distribution with mean 0.5, we can expect a edge size loss value near $0.5 * 500 * 0.05 = 1.25$, which is comparable to the other components. We execute the example script for explaining individual node 112 in Cora dataset and the result is shown in Figure 4.6. We can check that the value is near the predicted 1.25 and there is a decrease to 0.5. For the final value we have to take into account that irrelevant edges end up with a small value, but not zero, so they are responsible of a certain amount of the loss value. We can plot this outcome for the node 112 using a distribution plot of Seaborn (a histogram) for the values of the edge mask. The result is shown at Figure 4.7

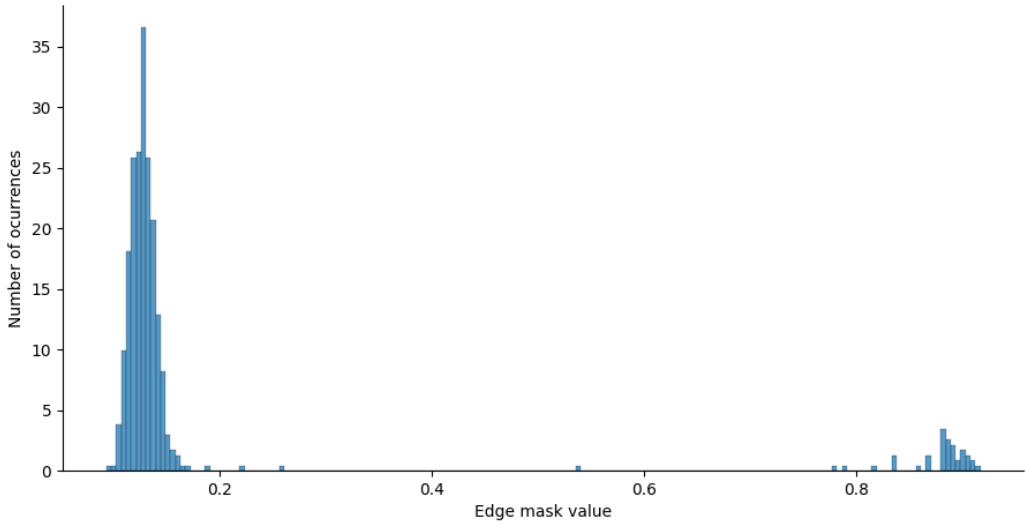

Figure 4.7: Distribution of edge mask value for node 112 at Cora with default settings (N=500)

The use of a fixed coefficient is mentioned in the original paper and is the default use for the implementation, but it suffers when the distribution of the N is wide. Looking at the distributions of the synthetic datasets from the original paper, we can say there is a small number of nodes with small N, as show in Figure 4.8 and Figure 4.9.

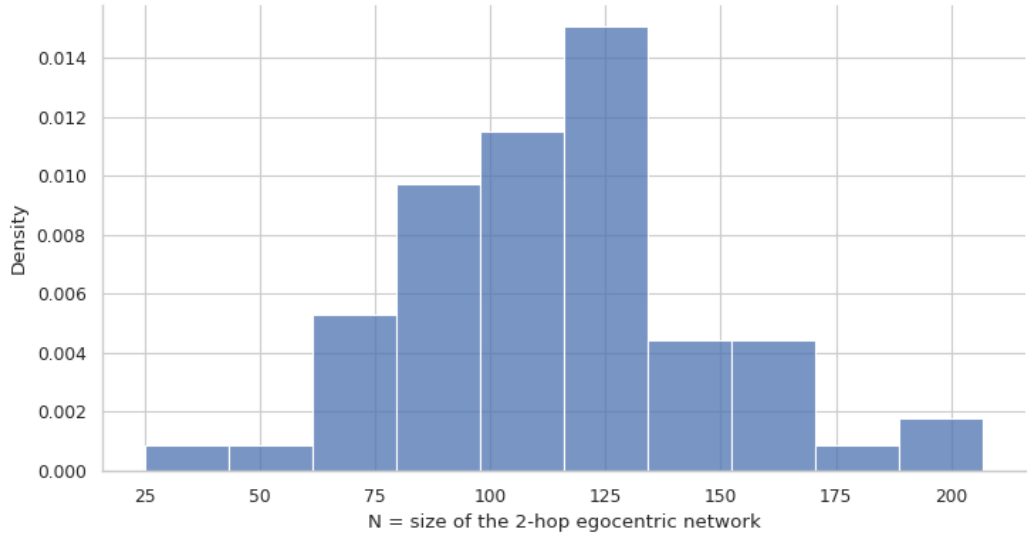

Figure 4.8: Distribution of N in the synthetic dataset number 1 from the original article

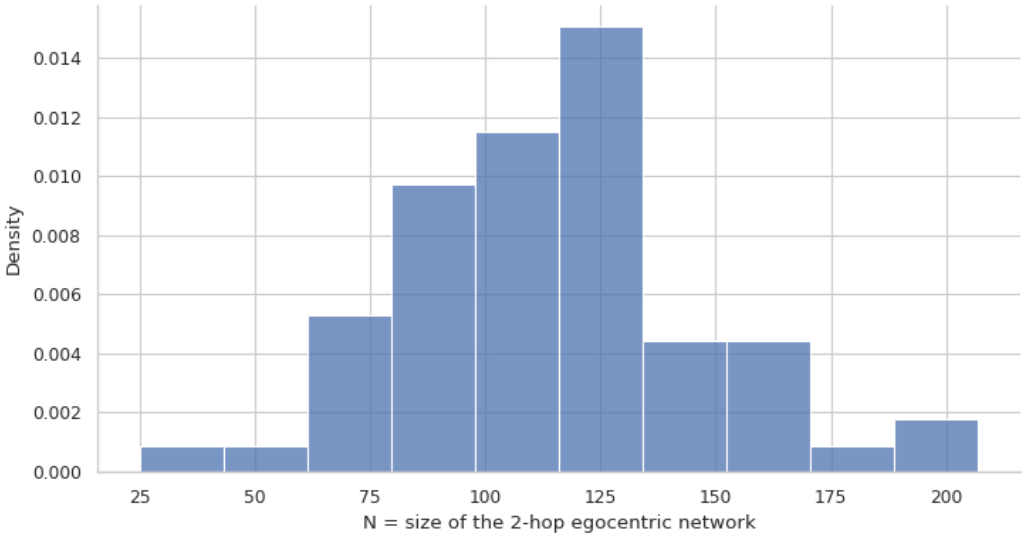

Figure 4.9: Distribution of N in the synthetic dataset number 2 from the original article

If we compare these distributions with the distribution of the classical citation datasets as Cora (Figure 4.10) or Pubmed 4.11) we can see that the difference is what is making more inconsistencies appear while explaining nodes at these datasets. Most of the nodes of these datasets present a value of N smaller than 100.

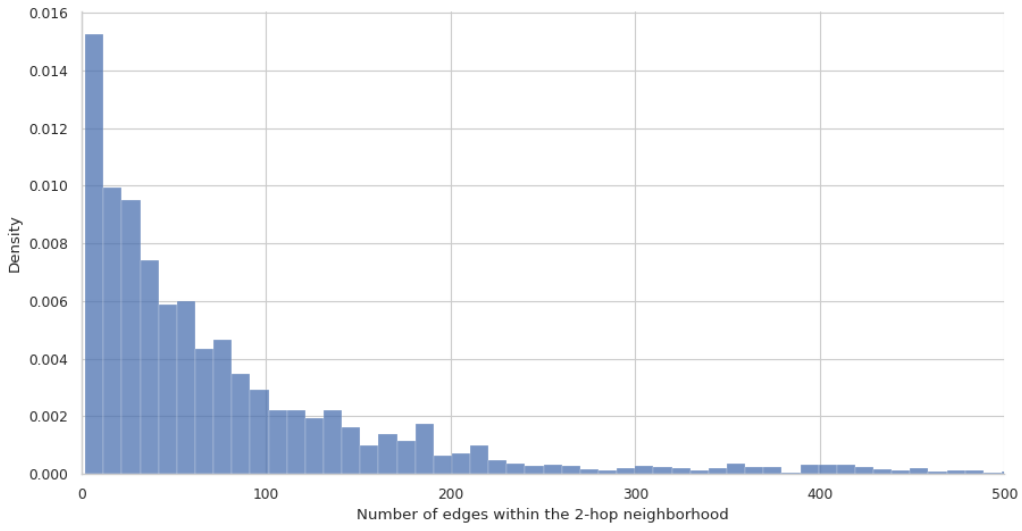

Figure 4.10: Distribution of N in the Cora dataset

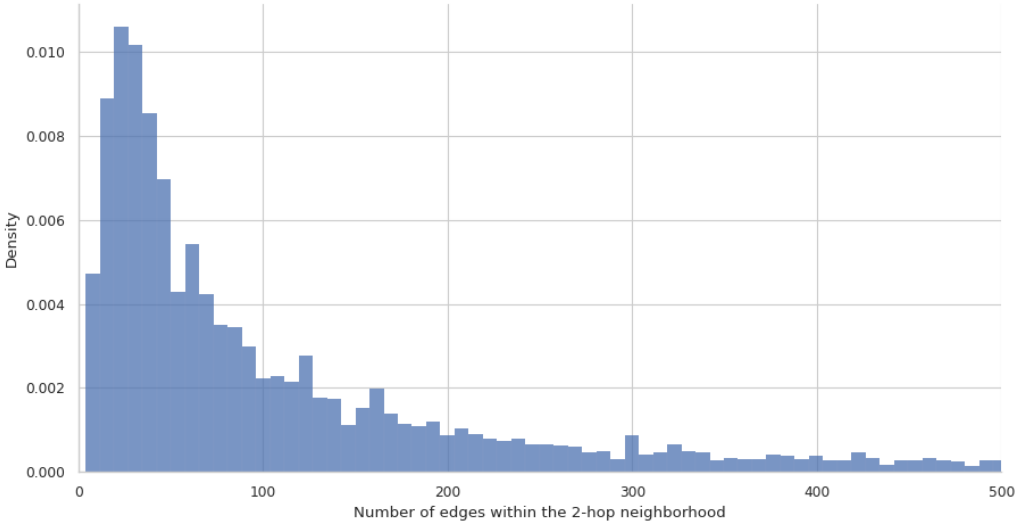

Figure 4.11: Distribution of N in the Pubmed dataset

In the next section we discuss how to choose an appropriate value for the coefficient edge_size to fulfill our objectives.

4.3 Adjusting the edge_size coefficient

How to decide a proper value for the coefficient edge_size depends on our goals. We can set it so the number of edges at the end of the training is around our objective and it does not maker other components irrelevant during the training. This means that the value of the coefficient edge_size can be divided into three regions:

- It does make the edge size component of the loss irrelevant
- It does other components irrelevant (particularly the prediction loss)
- In a equilibrium between these two regions

To set the edge_size coefficient we can take our desired value at the end of the training, which is composed by the term related to the relevant nodes and the term related to the irrelevant nodes. I The value of irrelevant edges is approximated to its conditional expected value ($E[X|X < 0.5]$) where X is the edge mask value, and we consider irrelevant any edge mask value under 0.5. Similarly, the value of relevant edges is approximated to its conditional expected value ($E[X|X \geq 0.5]$) where X is the edge mask value, and we consider relevant any edge mask value equal or above 0.5. Considering N the number of edges, S the number of edges for our explanation, we can compute the coefficient C as:

$$L = ([E[X|X < 0.5] * (N - S) + E[X|X \geq 0.5] * S]) * C$$

For simplification we extract the conditional expected values from our experiments and round the expectation to the first decimal place, obtaining $[E[X|X < 0.5] = 0.1$ and $E[X|X >= 0.5] = 0.9$. For $N >> S$ we can approximate the lower bound for L as $[E[X|X < 0.5] * N * C$. We can reorder the formula to obtain the coefficient:

$$C = \frac{10L}{N + 8S}$$

To introduce this modification in the code we need to compute N before calling the explainer. Also, in our experiments we see that the individual loss components end up with values around 0.5, so we can set this as an objective loss L to arrive at the following code:

Listing 4.4: Code for calling the explainer

```
node_idx = 10
S = 5 # explanation size
L = 0.5 # desired loss at the end of the training

_, _, _, hard_edge_mask = k_hop_subgraph(node_idx, 2, edge_index)
N = torch.sum(hard_edge_mask==True).item()
explainer = GNNExplainer(model, epochs=200)
explainer.coeffs['edge_size'] = 10*L/(N+8*S)
feat_maks, edge_mask = explainer.explain_node(node_idx, x, edge_index)
```

With this modification we repeat the experiment of Figure 4.4 to analyze the difference in relation to the instability of the explanations. The result is shown in Figure 4.12 and it confirms the improvement on this metric.

Moreover, using the Adjusted Coefficient produces more compact explanations, with smaller number of edges. A comparison is shown between Figure 4.13.

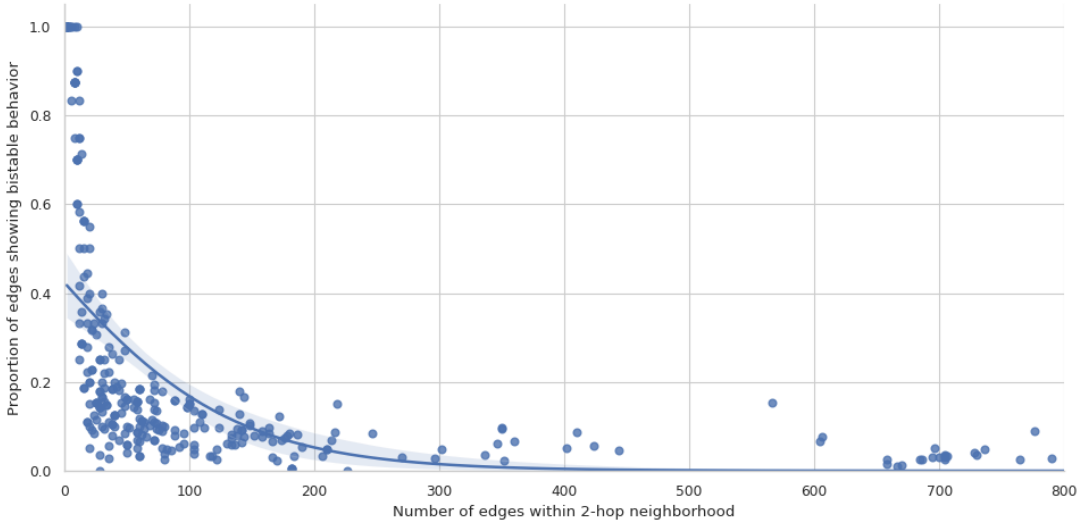

Figure 4.12: Proportion of bistable behaviour in edges regarding N with the adjusted edge_size coefficient (Cora dataset, 300 uniformly sampled nodes). Linear scale

```
[0.9103, 0.8305, 0.0631, 0.7547, 0.0736, 0.0740, 0.0717, 0.0749, 0.9186, 0.9592, 0.0490, 0.8682]
[0.9269, 0.8785, 0.0714, 0.8732, 0.0684, 0.0692, 0.9394, 0.9641, 0.8821, 0.0670, 0.0529, 0.7739]
[0.0364, 0.8383, 0.9277, 0.8577, 0.0637, 0.0709, 0.9372, 0.9428, 0.0404, 0.0677, 0.9278, 0.8978]
[0.9082, 0.8983, 0.0693, 0.8961, 0.9313, 0.9313, 0.0750, 0.0590, 0.9126, 0.9238, 0.0656, 0.8617]
[0.8800, 0.8025, 0.0618, 0.9010, 0.0729, 0.0656, 0.9312, 0.0741, 0.8824, 0.0659, 0.9246, 0.8525]
[0.9053, 0.8527, 0.0651, 0.8179, 0.9317, 0.9319, 0.9311, 0.9330, 0.9170, 0.0592, 0.0678, 0.8915]
[0.9167, 0.8516, 0.0682, 0.9054, 0.0596, 0.0710, 0.0742, 0.0737, 0.8834, 0.9211, 0.0657, 0.8176]
[0.8976, 0.9251, 0.0663, 0.8531, 0.9312, 0.0614, 0.0608, 0.0744, 0.9348, 0.0303, 0.0645, 0.8045]
[0.8539, 0.8696, 0.9285, 0.9197, 0.0612, 0.9327, 0.9425, 0.0698, 0.9328, 0.0650, 0.0664, 0.8624]
[0.9021, 0.9156, 0.0668, 0.9037, 0.0698, 0.0712, 0.9319, 0.0724, 0.9157, 0.0662, 0.0651, 0.8526]
```

Figure 4.14: 2-dimensional array of edge mask values for node 10 at Cora Dataset

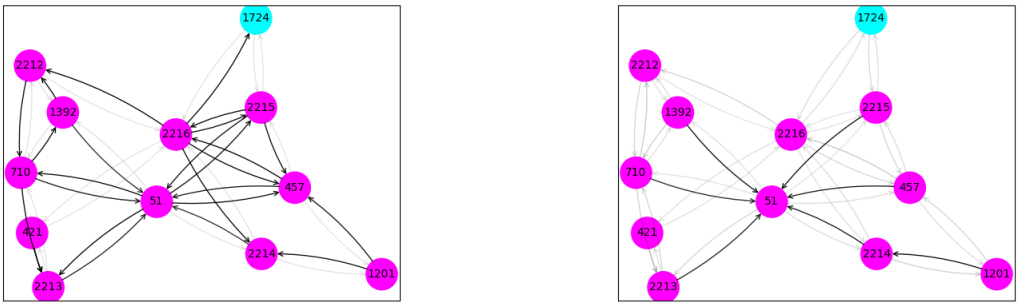

Figure 4.13: Explanation for node 51 at Cora Dataset. Left: Default Settings. Right: Adjusted Coefficient

Now the bistable behaviour of the edges is reduced, but it already appears. For handling this, in the next section we propose a solution based on the Monte Carlo method.

4.4 Using Monte Carlo to reduce instability

As we showed at the previous section, even after adjusting the edge_size coefficient, the bistable behaviour shows up. We are interested in the nodes that show consistently a stable behaviour.

We propose a Monte Carlo method, where we repeat the experiment and take a decision based on the outcome.

We wrap our code within a loop and store the edge mask list within a 2-dimensional array, where each row represent a Monte Carlo run. The columns continue representing the index of the node as in the original list.

For example, a Monte Carlo execution consisting on 10 runs for explaining node 10 at Cora dataset with original parameters (without using our adjusted edge_size method) generates the values shown at figure 4.14

Looking at the first edge (first column, representing the edge from node 10 to node 476) we can see that most values are over 0.9 but there is one value near zero, exhibiting the bistable behavior. Now if we look at the last edge (last column, representing the edge from node 2545 to node 10) we can see there is no bistable behaviour but all its values are under 0.9.

In the original implementation we would select the highest values that exceeds a certain threshold, but now we have not a single value but a set of values. We should decide on an aggregation policy. If we have to decide between both edges, different policies will produce different results as we can see at Table 4.1

In this work we will use the 0.9 Threshold count, being this a decision left to the user.

Policy	Edge 1	Edge 12
Mean	0.81374	**0.84827**
Trimmed Mean	**0.8967**	0.851375
Median	**0.9037**	0.85715
0.85 Threshold count	**9**	7
0.9 Threshold count	**6**	0

Table 4.1: Different policies selecting relevant edges

The following code acts as an example, where we use the sum of the number of elements through dimension 0 (columns) and then pick the top elements with torch.topk.

Listing 4.5: Code for calling the explainer with Monte Carlo method

```
threshold = 0.9
S = 5 # explanation size
edge_mask_mc = torch.zeros(mc_runs, edge_index.size(1)).to(device)

for run in range(mc_runs):
    explainer = GNNExplainer(model, epochs=200)
    feat_maks, edge_mask = explainer.explain_node(node_idx, x, edge_index)
    edge_mask_mc[run] = edge_mask

rel_edges = torch.topk((edge_mask_mc > threshold).sum(dim=0),S).indices.tolist()
```

4.5 Benchmarks of stability

Now that we have defined different methods to improve the stability of different nodes, we want to measure the impact of these improvements in different datasets. To do that, we will measure instability as the proportion of edges that show the bistable behaviour (it is relevant in some runs and irrelevant in others) for different combinations of the method.

This evaluation is executed with a fixed seed for reproducibility. The experiment is as follows:

1. We select a dataset
2. We select P random nodes for explanation
3. For each explanation we execute R consecutive runs
4. For each run we use four different methods: Baseline, Adjusted Coefficient (AC) , AC + Monte Carlo (MC) and log the result
5. We obtain a single value aggregating results

For step 5 we need to build a function that returns a single value for bench-marking. We design it to be bounded between 0 and 1, being 1 the maximum of stability for a selected dataset (across all the sampled nodes, across all the runs each edge is either relevant or irrelevant) and 0 the minimum (across all the sampled nodes, across all the runs each edge has a 50 percent of being relevant). This number is calculated for each edge and then the mean is returned. Then, the mean of every random node is aggregated to return the mean across all analyzed nodes, and this is the number we are using for our evaluation.

We can set the formula intuitively as:

$$e^i = \frac{min(e_0^i, e_1^i)}{max(e_0^i, e_1^i)}$$

Where e^i is the instability value for a certain edge i within a explanation, e_0^i is the number of times the edge is considered irrelevant across the R runs and e_1^i is the number of times the edge is considered relevant across the R runs and e_1^i. It can also be computed without using min/max functions by using the following equation:

$$e^i = 1 - \frac{|(\sum_{j=1}^{R} e_j) - \frac{R}{2}|}{\frac{R}{2}}$$

Where e^i is the instability value for a certain edge i within a explanation and e_j is the discretized edge mask value (0 or 1) at the j-th run for edge i.

The stability of a certain node is the mean of the instability of its edges (N is the number of edges within the 2-hop neighborhood):

$$n^j = \frac{1}{N} \sum_{i=1}^{N} e^i$$

And finally the stability for a sample of nodes is 1 minus the mean of the instability of the sampled nodes:

$$s = 1 - \frac{1}{P} \sum_{j=1}^{P} n^j$$

Stability (95% CL, 5% ME)			
Model+Dataset	Baseline	Adjusted Coefficient (AC)	Monte Carlo (MC) + AC
GCN+Cora	0.796	0.918	**0.924**
GCN+Pubmed	0.818	0.970	**0.974**
GAT+Cora	0.771	0.916	**0.918**
GAT+Pubmed	0.784	0.975	**0.979**

Table 4.2: Evaluation of stability

The major drawback of our proposal is the execution time, which is increased multiple times (increases linearly with the number of runs in the Monte Carlo extension). While a ten-fold increase may seem problematic, for a single prediction the execution time is still under 30 seconds in an average laptop. For our simulations, involving 300 nodes per dataset, it is indeed an long process that must be scheduled.

We have restricted ourselves to modify the minimum regarding the internals of GNNExplainer, but modifying the learning rate and number of epochs may be useful to reduce the execution time. Another way of reducing execution time could be using Julia, which, as stated in [46], is typically faster than Python for execution of Machine Learning techniques. Porting GNNExplainer to Julia, particularly to its package GeometricFlux, is an interesting next step.

While the best results are obtained with the Adjusted Coefficient combined with Monte Carlo, the improvement over Adjusted Coefficient is not enough to justify the increase in execution time, therefore we recommend using only the Adjusted Coefficient.

We can now compare the distribution between the GCN+Cora with default settings and with our proposed Adjusted Coefficient (AC) as shown in Figure 4.15

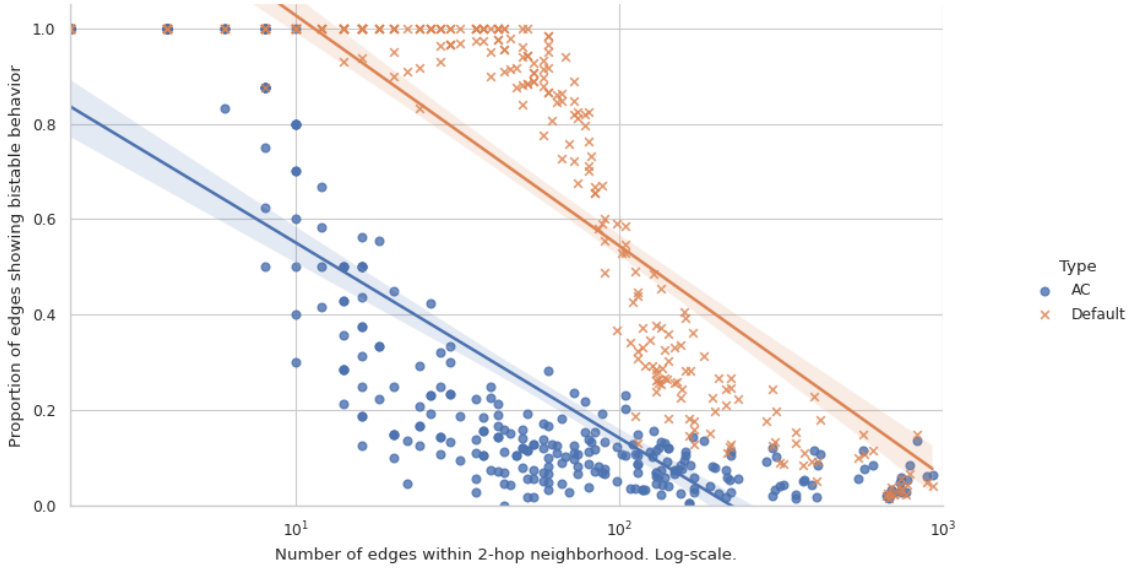

Figure 4.15: Comparison of the proportion of bistable behaviour in edges before and after coefficient adjustment (Cora dataset, 300 uniformly sampled nodes per type) Log-scale in the horizontal axis

4.6 Improving the visualization of GNNExplainer

We have shown that the explanation size should remain around 7, so it is compact enough to be understood by a regular person. However, N, the number of edges that must be analyzed, can be in the order of the hundreds or thousands. With the current visualization tool included in the Pytorch Geometric package, all of theses non-relevant edges are plotted with grey lines to contrast with the relevant edges, which are plotted in black. We can see an example at Figure 4.16.

If we try to analyze the graphical presentation of the explanation of figure we will have difficulties because the nodes are overlapped and there are no clear patterns of relationship. This happens when the number of nodes in the 2-hop network is over 20 nodes. Note that even selecting a threshold to show less edges with color black, all the nodes keep appearing.

Our proposal to improve the visualization to show only these nodes that are the origin or destination of a relevant edge. To draw only selected nodes the method "draw_networkx_nodes" allows a parameter "selected" where a list of nodes can be determined. Additionally, the explained node is highlighted with a black border. Resulting plots are shown in Figure 4.16 and Figure 4.17.

As a counterexample, in Figure 4.18 and Figure 4.19 we can see that being the size of the computation graph not as big as in the previous example, the current visualization function is working properly, and it shows more information than our proposal.

We recommend to use a combination of the functions, depending upon the size of the computation graph of the node being explained.

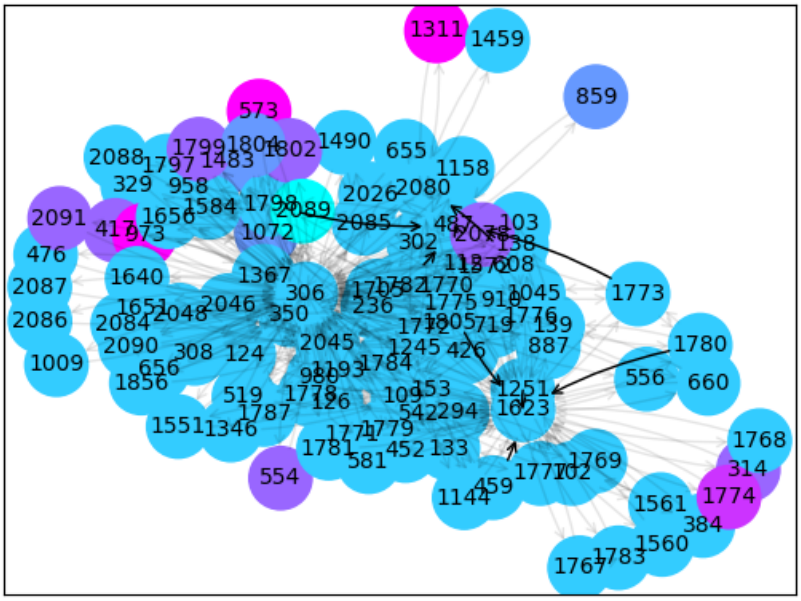

Figure 4.16: Default explanation for node 112 at Cora with default settings. Node colors represent different labels. Edge colors represent the relevance of the edge.

Figure 4.17: Our proposed explanation for node 112 at Cora. Node colors represent different labels. Edge colors represent the label of the source node.

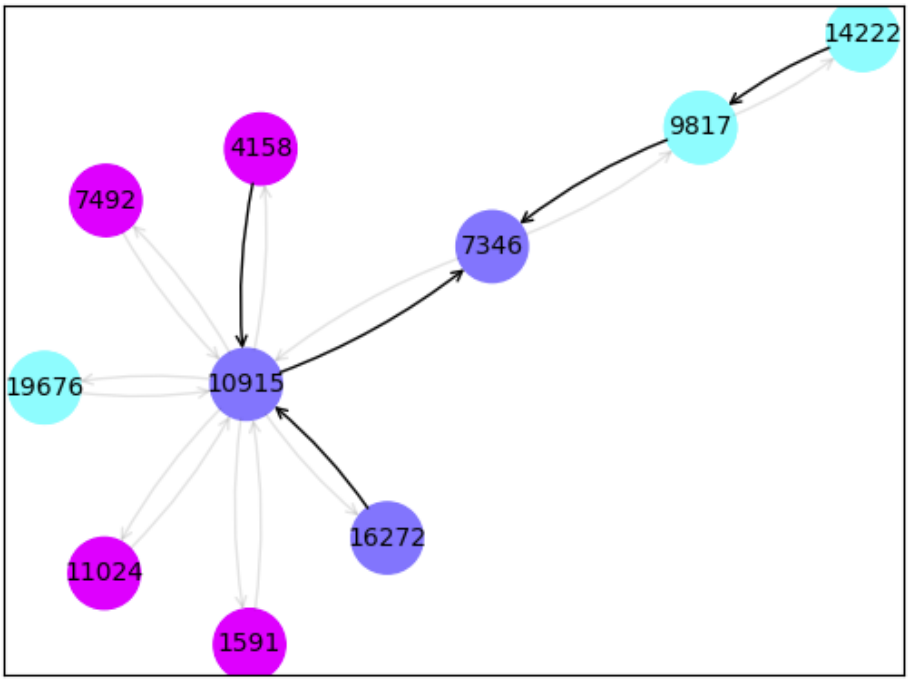

Figure 4.18: Default explanation for node 7364 at Pubmed. Node colors represent different labels. Edge colors represent the relevance of the edge

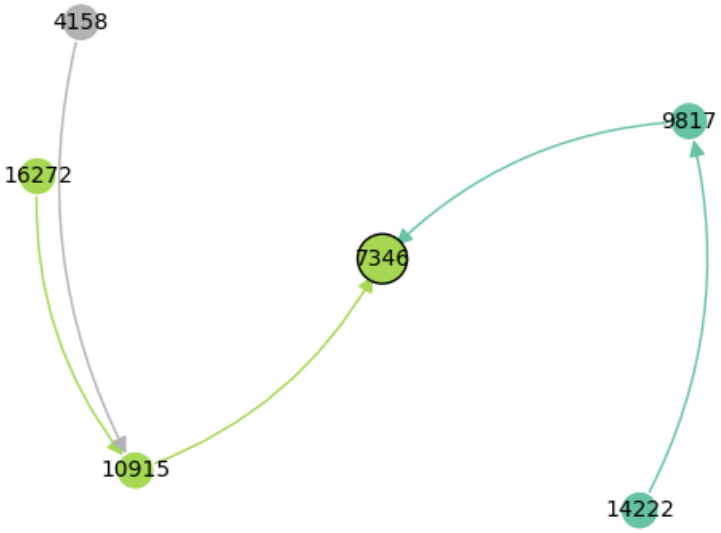

Figure 4.19: Our proposed explanation for node 7364 at Pubmed. Node colors represent different labels. Edge colors represent the label of the source node

Listing 4.6: Graphical explanation function

```
def graphical_explanation(G, y, node_idx):
    try:
```

```
        node = list(G.nodes).index(node_idx)
    except Exception:
        G.add_node(node_idx)
        node = list(G.nodes).index(node_idx)

    edge_colors = []
    for edge in list(G.edges):
        edge_colors.append(y[edge[0]].item())

    node_edge_colors = ['white' for x in y]
    node_edge_colors[node] = 'black'

    node_sizes = [300 for x in list(G.nodes)]
    node_sizes[node] = 500

    node_colors = [y[x] for x in list(G.nodes)]

    fixed_positions = {node_idx: (0, 0)}
    fixed_nodes = fixed_positions.keys()
    pos = nx.spring_layout(G, pos=fixed_positions,
                           fixed=fixed_nodes,
                           center=[0, 0],
                           seed=1)
    nx.draw_networkx_nodes(G, pos, node_size=node_sizes,
                           node_color=node_colors,
                           edgecolors=node_edge_colors,
                           cmap=plt.cm.Set2)
    nx.draw_networkx_labels(G, pos, font_size=10)
    nx.draw_networkx_edges(G, pos, edge_color=edge_colors,
                           edge_cmap=plt.cm.Set2,
                           connectionstyle="arc3,rad=0.2",
                           arrowstyle='-|>', arrowsize=15)
    plt.axis('off')
    ax = plt.gca()

    return ax
```

The graphical_explanation function at Listing 4.6 receives the graph G for explanation, the vector y with the labels of the nodes and the node_idx with the index of the node being explained, and return a Matplotlib axis for plotting the explanation. Note that the input is the networkX Graph reduced to the nodes with relevant edges, so taking the vector rel_edges as the reference containing the indexes of the relevant edges, we first have to build the Graph as shown in the following listing:

Listing 4.7: Code for preparing the explanation sub-graph

```
G = nx.DiGraph()

for edge in rel_edges:
    G.add_edge(edge_index[0][edge].item(), edge_index[1][edge].item())
```

The improvement of the visualization can not be easily quantified, but we can make use of a similar concept to Tufte's Data-Ink Ratio from [47]. We compare the proportion of non-white pixels between the original plot and our proposal for the same explanation. Following Tufte's ideas, for the same explanation a smaller number of non-white pixels is better. For computing a similar concept we can take the size of the explanation and the proportion of non white pixels in the image. However, this metric is not enough to tell if a visualization is better than the other, it is up to the user to decide which plot fits better her goals.

The result can be compared with the same explained node (Figure 4.20)

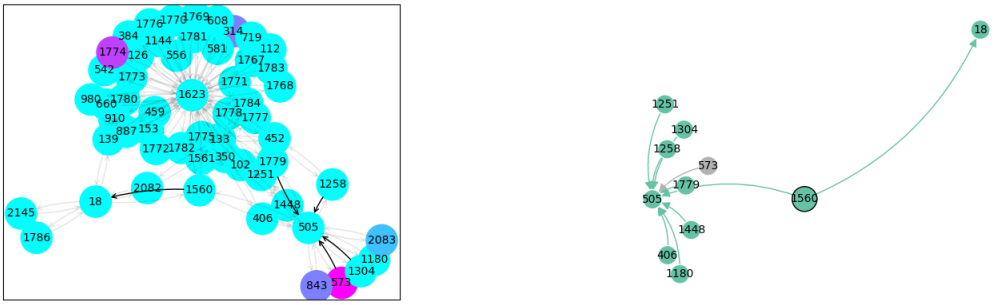

Figure 4.20: Explanation for node 1560 at Cora Dataset. Left: Default visualization. Right: Proposed visualization

The experiment is as follows:

1. We select a dataset
2. We select P random nodes for explanation
3. For each explanation we create the original plot and our proposed plot
4. The proportion of white pixels is computed for both plots
5. The average proportion is returned

Proportion of white pixels (95% CL, 5% ME)		
Dataset	Baseline	Proposal
Cora	85.1 %	**98.1 %**
Pubmed	79.9 %	**97.9 %**

Table 4.3: Evaluation of proportion of white pixels

Chapter 5

Conclusions

The concept of explainability or interpretability is gaining much attention on the academic and industrial community of Machine Learning. Its application to Graph Neural Networks is, however, already starting, with the seminal paper being published in 2019. As GNNs turn more popular, we expect this topic will became crucial.

In particular, we consider that using black-box models for decisions that affects the life of people is not ethical, and therefore Explainable Artificial Intelligence is not an option, but a moral imperative. We are particularly interested in the application of Graph Machine Learning to Computation Social Sciences, therefore our motivation for the topic of this work is related to the tenets of a democratic society.

The tool for describing a prediction is called an explainer. When related to an individual instance explanation, it receives the graph and the set of labels, plus the prediction of a node, and it must assess why the prediction took place.

In this work we have explored the state of the art of explainers for Graph Neural Networks. The first explainer to be published was GNNExplainer and is commonly referenced as the *de facto* standard. We analyze the explainers with a published implementation, namely GNNExplainer, PG-Explainer, COGE, PGM-Explainer and GraphLIME. A relevant field of study that many implementations use is the Information Theory field, as they rely on the concepts of cross-entropy and mutual information. Bayesian principles are also used, as in the case of PGM-Explainer.

We select GNNExplainer as the tool for this work because it is the only one with an available implementation on a major Data Science software package, Pytorch. However we have found that this implementation suffers a series of issues when being applied to common datasets of academic citations, which are typically used for bench-marking (Cora and Pubmed). The three main issues are:

1. lack of stability of the explanations among different runs of the same experiment
2. excessive number of edges for a single explanation
3. difficult to visually interpret plots

Regarding the lack of stability we analyze the origin of the randomness and find that it lies within a function that sets the initial masks for edges and features. This mask is constructed with a Gaussian distribution, and the formula for setting it is related to the value N, the number of edges in the computation graph. In summary, this value N is used for the computation of:

1. the standard deviation of the Gaussian random mask used at the initialization

2. the component of the loss function related to the size of the explanation (number of relevant edges)

After a detailed analysis of the algorithm we have determined that the distribution of the number of edges analyzed per explanation is quite different in these citation datasets. One of the differences between the citation datasets and the datasets used in the original GNNExplainer article is the distribution of the size of the computation graph. In the citation datasets, most of the nodes have a size of computation graph below 100, whereas in the datasets of the original article the mean value is over 100. This discrepancy makes that the use of fixed parameters coming from the original article does not fit all the scenarios.

We have analyzed the impact of this distribution in terms of stability (how similar are the explanations for similar instances) and in terms of the contribution to the loss function, and we have confirmed that the fixed coefficient limits the contribution to the loss function to a subset of the node explanations, those with the number of edges within a range.

To address the aforementioned issue, we designed an Adjusted Coefficient computed before the explanation to make the contribution to the loss function within our objective. We showed that this modification improves substantially the stability of the explanation by conducting an experiment. The experiment consists on explaining the same node multiple times, and analyzing the difference in the multiple explanations. A high stability means that most of the explanations should be identical.

We have also included a Monte Carlo method to increase stability, but the improvement is small and the cost in terms of execution time is high, therefore the this approach still require some work to reduce temporal complexity.

Finally, we have addressed the graphical explanation produced by the current Pytorch's implementation, which suffers from overlapping nodes and edges when the size of the neighborhood is large (i.e. more than 50 edges). We propose showing only the nodes that are connected to relevant edges and using color at the edges for describing the source node. Our goal is to have pictures with less coloured pixels (as ink in Tufte's ideas) and allow for an easier interpretation of the plot.

Summarizing, our main contributions are:

1. a state of the art of explainers for Graph Neural Networks
2. a deep analysis of GNNExplainer regarding the stability issues
3. a variable value for one of the parameters, adjusting it to the size of the node's neighborhood
4. a benchmark with different GNNs and datasets
5. a new visualization tool showing only the nodes concerned within the explanation

With this contributions we improve the stability of the baseline method by more than 10% in experiments using Graph Convolutional Networks (GCN) and Graph Attention Networks (GAT) with datasets Cora and Pubmed.

We have employed only minor modifications in the code and we are essentially using the same principles as GNNExplainer, so the mathematical foundations are the same as the original article.

5.1 Drawbacks

We have designed a Monte Carlo method to improve stability, but the linear increase in execution time makes it not worthy because it shows little improvement over the Adjusted Coefficient method. Also, this method needs an aggregation function that adds complexity to the analysis of the method.

The visualization function proposed improves interpretability for explanations where the computation graph size is over 30 edges, but in the rest of the cases, the current visualization function works better because it gives more information to the user without interfering with the interpretability.

5.2 Future work

GNNExplainer is the first method for explainability for GNNs and has been used as a baseline for other methods. It is only implemented in Python, so we plan to contribute a porting to Julia, in particular to the GeometricFlux package.

Regarding the method, we will research about methods that construct compact explanations with only one connected component, as we presented in the background chapter.

Beyond the Information Theory concepts used in GNNExplainer, we are interested in developing a tool with the ideas of the PGM-Explainer article, and explore the new field of Probabilistic Machine Learning [48]. In general, we are interested in applying Bayesian principles to deep learning and learn how it facilitates the interpretability of black-box models.

Appendix A

Reproducibility instructions

For reproducing the results in this work, the base source code is located in a public repository [1]. The snippets of code provided must be used within this example.

The code should be executed using Python 3.8 and the most relevant packages and its versions are:

```
- cudatoolkit==10.1.243
- keras-preprocessing==1.1.0
- numpy==1.19.2
- numpy-base==1.19.2
- pytorch==1.7.0
- statsmodels==0.12.1
- tensorboard==2.4.0
- tensorboard-plugin-wit==1.7.0
- tensorflow==2.3.0
- tensorflow-base==2.3.0
- tensorflow-estimator==2.3.0
- torchvision==0.8.1
- littleballoffur==2.1.7
- matplotlib==3.3.3
- networkit==7.1
- networkx==2.5
- opencv-python==4.4.0.46
- scikit-learn==0.23.2
- scipy==1.5.4
- seaborn==0.11.0
- tensorboardx==2.1
- torch-cluster==1.5.8
- torch-geometric==1.6.3
- torch-scatter==2.0.5
- torch-sparse==0.6.8
- torch-spline-conv==1.2.0
- tqdm==4.54.0
```

In particular, for the installation of Pytorch geometric under a GPU, we have followed the instructions of Pytorch Geometric documentation [2]:

```
conda install pytorch=1.7 torchvision cudatoolkit=10.1 -c pytorch
```

[1] https://github.com/rusty1s/pytorch_geometric/blob/master/examples/gnn_explainer.py
[2] Pytorch Geometric: https://pytorch-geometric.readthedocs.io/en/latest/notes/installation.html

```
pip install torch-scatter==latest+cu101 -f https://pytorch-geometric.com/whl/torch-1.7.0.html
pip install torch-scatter -f https://pytorch-geometric.com/whl/torch-1.7.0+cu101.html
pip install torch-sparse -f https://pytorch-geometric.com/whl/torch-1.7.0+cu101.html
pip install torch-cluster -f https://pytorch-geometric.com/whl/torch-1.7.0+cu101.html
pip install torch-spline-conv -f https://pytorch-geometric.com/whl/torch-1.7.0+cu101.html
pip install torch-geometric
```

The host system is running Ubuntu 20.04 on a HP All in One PC (27-XA0004NS) with a Geforce MX-130 GPU.

Appendix B

Node sampling

During our previous quantitative analysis some of the tests were executed during long periods of time. For the evaluation of the selected datasets this time is affordable by using Cloud services, but it raises the concern on how to proceed with real datasets. Analyzing only a fraction of the nodes, while keeping a reasonable approximation to the true values, needs a careful selection of the nodes that must ensure that some properties are maintained in the sample population. This problem is known as Graph Sampling.

In our particular case, the goal is getting representative subset of vertices from a target population that can be sampled directly. This is the most common use of graph sampling in sociology studies, according to [49]. This means that we do not need the output to be a sub-graph (with nodes and edges) that will be used afterwards. In our setup the same original graph will be used, but the testing will be limited to the set of representative nodes. Therefore, we will discard the graph information and we will approach the problem as sampling a set.

Our goal is that the distribution of sizes of the egocentric network for a given k in the sampled set maintains a similar distribution of the original dataset.

To check whether the sampling is working we use the distance formula on previous section. Also, we can plot the histogram. There is a method in the networkX library [50] for computing the k-hop egocentric network (nx.ego_graph) but the internal verification make it slow for our purpose. Instead, we look for the specific part that applies to our case inside its code and use it directly (nx.single_source_shortest_path_length).

Listing B.1: Code for obtaining the number of edges in the k-hop egocentric network

```
# Option 1 with list comprehension (for compact code)
ego_edges = [len(nx.ego_graph(G, node, radius=k).edges)
for node in G.nodes()]

# Option 2 with for loop (for readability)
ego_edges = [0] * len(G.nodes())
for n in G.nodes():
    node_list = list(nx.single_source_shortest_path_length(G,
    n, cutoff=k).keys())
    H = G.subgraph(node_list)
    ego_edges[n] = (len(H.edges))

# Option 3 with list comprehension (for speed)

ego_edges = [len(G.subgraph(list(nx.single_source_shortest_path_length(
                G, node,cutoff=k).keys())).edges) for node in nodes]
```

Now that we have the list with the number of edges within the k-hop egocentric network,

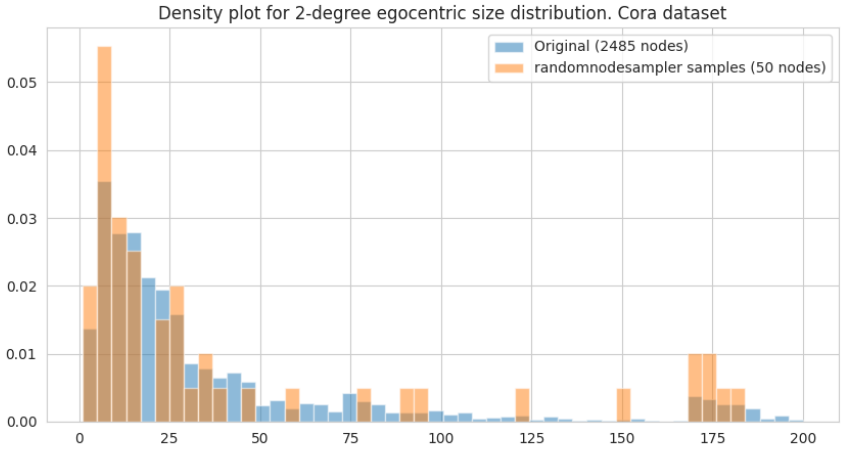

Figure B.1: A comparison of the histograms of egocentric networks (k=2) for all the nodes and the selected representatives (for sizes up to 200)

we can plot the histogram.

Using the open source library LittleBallOfFur [51] we can call different samplers from a catalog (RandomNodeSampler, DegreeBasedSampler, RandomWalkSampler and many others).

From the list of node sampling algorithms, we can pick the DegreeBasedSampler and modify it to match our objective.

Listing B.2: Code for DegreeBasedSampler in LittleBallOFFur

```
def _create_initial_node_set(self, graph: Union[NXGraph, NKGraph])
-> List[int]:
    """
    Choosing initial nodes.
    """
    nodes = [node for node in
    range(self.backend.get_number_of_nodes(graph))]
    degrees = [float(self.backend.get_degree(graph, node))
    for node in nodes]
    degree_sum = sum(degrees)
    degrees = [degree/degree_sum for degree in degrees]
    sampled_nodes = np.random.choice(nodes, size=self.number_of_nodes,
                                     replace=False, p=degrees)
    return sampled_nodes
```

We can compare the histograms of our proposal and the original set. A function must be computed for comparison, which should be decided based on the goals of the developer. Here we show a reduced candidate list based on [52]:

- Hellinger Distance: $D_{L0} = \sum_{i=1}^{n} h1_i \neq h2_i$
- Manhattan Distance: $D_{L1} = \sum_{i=1}^{n} |h1_i - h2_i|$
- Euclidean Distance: $D_{L2} = sqrt \sum_{i=1}^{n} (h1_i - h2_i)^2$
- Chybyshev Distance $D_{L_\infty} = max_i |h1_i - h2_i|$

- Cosine Distance: $D_{CO} = 1 - \sum_{i=1}^{n} h1_i h2_i$
- Pearson Correlation: $D_{CR} = \frac{\sum_{i=1}^{n}(h1_i - 1/n)(h2_i - 1/n)}{\sqrt{\sum_{i=1}^{n}(h1_i - 1/n)^2 (h2_i - 1/n)^2}}$

where n is the number of bins, $h1_i$ is the i-th value for the histogram of the full graph and $h2_i$ the i-th value for the histogram of the sampled graph.

After comparing with the various metrics, we find the simple uniform sampling will perform reasonably well for most situations and we will use it as our sampling algorithm. We will use a sample size of P=300 nodes, which is over 95% of confidence-level with a 5% error margin for both Cora and Pubmed datasets, based on the Normal approximation (using the Central Limit Theorem, as we are interested in the mean of the stability value across all nodes).

Bibliography

[1] Christoph Molnar. *Limitations of Interpretable Machine Learning Methods.* Molnar, Christoph, 2019.

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

[3] Alex Fout, Jonathon Byrd, Basir Shariat, and Asa Ben-Hur. Protein interface prediction using graph convolutional networks. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6533–6542, Red Hook, NY, USA, 2017. Curran Associates Inc.

[4] Michael M. Bronstein, Joan Bruna, Yann LeCun, Arthur Szlam, and Pierre Vandergheynst. Geometric deep learning: Going beyond euclidean data. *IEEE Signal Processing Magazine*, 34(4):18–42, Jul 2017.

[5] William L. Hamilton. Graph representation learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 14(3):1–159, 2020.

[6] Ines Chami, Sami Abu-El-Haija, Bryan Perozzi, Christopher Ré, and Kevin Murphy. Machine learning on graphs: A model and comprehensive taxonomy, 2021.

[7] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk. *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, Aug 2014.

[8] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks, 2016.

[9] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009.

[10] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives, 2014.

[11] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[12] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks, 2018.

[13] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

[14] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate, 2016.

[15] Cynthia Rudin. Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead, 2019.

[16] Tim Miller. Explanation in artificial intelligence: Insights from the social sciences. *Artificial Intelligence*, 267:1–38, 2019.

[17] Marcel Boumans. *The Difference Between Answering a 'Why' Question and Answering a 'How Much' Question*, pages 107–124. Springer Netherlands, Dordrecht, 2006.

[18] Peter Lipton. Contrastive explanation. *Royal Institute of Philosophy Supplement*, 27:247–266, 1990.

[19] Hao Yuan, Haiyang Yu, Shurui Gui, and Shuiwang Ji. Explainability in graph neural networks: A taxonomic survey, 2020.

[20] Diogo V. Carvalho, Eduardo M. Pereira, and Jaime S. Cardoso. Machine learning interpretability: A survey on methods and metrics. *Electronics*, 8(8):832, Jul 2019.

[21] Rex Ying, Dylan Bourgeois, Jiaxuan You, Marinka Zitnik, and Jure Leskovec. Gnnexplainer: Generating explanations for graph neural networks, 2019.

[22] Hao Yuan, Jiliang Tang, Xia Hu, and Shuiwang Ji. Xgnn: Towards model-level explanations of graph neural networks. *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, Jul 2020.

[23] Dongsheng Luo, Wei Cheng, Dongkuan Xu, Wenchao Yu, Bo Zong, Haifeng Chen, and Xiang Zhang. Parameterized explainer for graph neural network, 2020.

[24] Minh N. Vu and My T. Thai. Pgm-explainer: Probabilistic graphical model explanations for graph neural networks, 2020.

[25] Lukas Faber, Amin K. Moghaddam, and Roger Wattenhofer. Contrastive graph neural network explanation, 2020.

[26] Qiang Huang, Makoto Yamada, Yuan Tian, Dinesh Singh, Dawei Yin, and Yi Chang. Graphlime: Local interpretable model explanations for graph neural networks, 2020.

[27] Alexander B Wiltschko, Benjamin Sanchez-Lengeling, Brian Lee, Emily Reif, Jennifer Wei, Kevin James McCloskey, Lucy Colwell, Wesley Qian, and Yiliu Wang. Evaluating attribution for graph neural networks. In *Advances in Neural Information Processing Systems 33*, 2020. https://papers.nips.cc/paper/2020/hash/417fbbf2e9d5a28a855a11894b2e795a-Abstract.html.

[28] Shangsheng Xie and Mingming Lu. Interpreting and understanding graph convolutional neural network using gradient-based attribution method, 2019.

[29] Xiaoxiao Li and Joao Saude. Explain graph neural networks to understand weighted graph features in node classification, 2020.

[30] Thomas Schnake, Oliver Eberle, Jonas Lederer, Shinichi Nakajima, Kristof T. Schütt, Klaus-Robert Müller, and Grégoire Montavon. Higher-order explanations of graph neural networks via relevant walks, 2020.

[31] Yue Zhang, David Defazio, and Arti Ramesh. Relex: A model-agnostic relational model explainer, 2020.

[32] P. E. Pope, S. Kolouri, M. Rostami, C. E. Martin, and H. Hoffmann. Explainability methods for graph convolutional neural networks. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 10764–10773, 2019.

[33] Federico Baldassarre and Hossein Azizpour. Explainability techniques for graph convolutional networks, 2019.

[34] Chaojie Ji, Ruxin Wang, and Hongyan Wu. Perturb more, trap more: Understanding behaviors of graph neural networks, 2020.

[35] Michael Sejr Schlichtkrull, Nicola De Cao, and Ivan Titov. Interpreting graph neural networks for nlp with differentiable edge masking, 2020.

[36] Fei Wang, Daniel Zheng, James Decker, Xilun Wu, Grégory M. Essertel, and Tiark Rompf. Demystifying differentiable programming: Shift/reset the penultimate backpropagator, 2019.

[37] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[38] Alan E. Gelfand and Adrian F. M. Smith. Sampling-based approaches to calculating marginal densities. *Journal of the American Statistical Association*, 85(410):398–409, 1990.

[39] Andrew Kachites McCallum, Kamal Nigam, Jason Rennie, and Kristie Seymore. Automating the construction of internet portals with machine learning. *Inf. Retr.*, 3(2):127–163, July 2000.

[40] Galileo Namata, Ben London, Lise Getoor, and Bert Huang. Query-driven active surveying for collective classification. In *ICML Workshop on MLG*, 2012.

[41] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. *Mining of Massive Datasets.* Cambridge University Press, USA, 2nd edition, 2014.

[42] Alex Bavelas. Communication Patterns in Task-Oriented Groups. *Acoustical Society of America Journal*, 22(6):725, January 1950.

[43] Pascal Danscoine and Karl Kansky. Measures of network structure, 1989.

[44] M. Duggan. Social media update 2014, 2015.

[45] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, page 807–814, Madison, WI, USA, 2010. Omnipress.

[46] Viktor Axillus. *Comparing Julia and Python: An investigation of the performance on image processing with deep neural networks and classification.* PhD thesis, Blekinge Institute of Technology, Faculty of Computing, Department of Software Engineering., 2020.

[47] Edward R. Tufte. *The Visual Display of Quantitative Information.* Graphics Press, Cheshire, CT, 2 edition, 2001.

[48] Kevin P. Murphy. *Probabilistic Machine Learning: An introduction.* MIT Press, 2021.

[49] Pili Hu and Wing Cheong Lau. A survey and taxonomy of graph sampling. *CoRR*, abs/1308.5865, 2013.

[50] Aric Hagberg, Pieter Swart, and Daniel Chult. Exploring network structure, dynamics, and function using networkx, 01 2008.

[51] Benedek Rozemberczki, Oliver Kiss, and Rik Sarkar. Little Ball of Fur: A Python Library for Graph Sampling. In *Proceedings of the 29th ACM International Conference on Information and Knowledge Management (CIKM '20)*, page 3133–3140. ACM, 2020.

[52] K. Meshgi and S. Ishii. Expanding histogram of colors with gridding to improve tracking accuracy. In *2015 14th IAPR International Conference on Machine Vision Applications (MVA)*, pages 475–479, 2015.