Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Learning From Networks Final Report

Better quality embeddings for node classification combining node classification with link prediction in Graph Neural Networks.

Authors: Formaggio Alberto, Bedin Manuel, Zebele Daniele

1 Abstract

In this research paper, we explore the relationship between node classification and link prediction within the context of graph neural networks (GNNs). Our study aims to assess whether the fusion of these two tasks within a graph can yield advantages over employing GNNs solely for node classification. By integrating the predictive capabilities of link prediction with the node classification capabilities of GNNs, we seek to enhance the overall performance and effectiveness of graph-based models.

The code used in this experiment is publicly available at `https://github.com/AlbertoFormaggio1/fine_tuning_classification_prediction_GNN`

2 Introduction

Deep Learning approaches have had a high impact in recent years on various problems, resulting in very good performance. Classic deep learning tasks involve data that can be mapped to the Euclidean space (e.g. images, text, etc.).

However, the same approaches cannot be applied in many other applications where data representation is non-Euclidean, like graphs. Over the last few years, several approaches have been used to learn from networks, and among others, Graph Neural Networks (GNN) have resulted in significant success. GNNs can obtain graph embeddings in real-world applications, such as classification and link prediction.

Historically, classification and link prediction have been addressed independently from one another, with efforts to improve performance. However, we believe these two tasks may be interconnected, and that training on one task could potentially enhance the results of the other. In our case, we focused on using link prediction to improve the classification accuracy. This idea has already been explored previously [1], leading to some improvements in performance, even though they did not exploit the potential of generalization that GNNs can offer.

In this paper, we aim to investigate a fundamental question: Can we enhance the quality and generality of embeddings created by training a GNN on multiple tasks? While the Multi-Task Graph Convolutional Network (MTGCN) [2] has already ventured into this territory, its primary objective was to address multiple tasks within a single network, whereas our focus is solely on improving classification performance.

3 Method

3.1 Computational Problem

We will focus on optimizing the training process across multiple tasks to achieve improved performance in classification.

Given the input graph $G = (V, E)$, where n is the number of nodes in the graph, f is the feature dimension of the nodes, and k is the dimensionality of each node embedding, G can be represented with the features matrix $\mathbf{X} \in \mathbb{R}^{f \times n}$ (in which each column corresponds to a node), the adjacency matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$, and the degree matrix $\mathbf{D} \in \mathbb{R}^{n \times n}$. We will use a graph neural network to process the input features $\mathbf{X}$ and the normalized adjacency matrix $\tilde{\mathbf{A}}$ [3] to generate the output embeddings $\mathbf{Z} \in \mathbb{R}^{k \times n}$, where each column of Z corresponds to the embedding of a node of G.

Recent papers [4] have found that oversmoothing increases exponentially when the number of layers employed increases. Furthermore, for downstream tasks, vanilla GNNs can outperform deeper and more complex models [5]. For this reason, we decided to set the number of GNN layers to $j = 2$.

Let W^j_{Ti} be the weight matrices used for processing $\mathbf{Z}$, where $T \in \{C, P\}$ indicates whether the linear layer is for the classification (C) or link prediction task (P), i is the training iteration, and j is the depth of the layer. For instance, $\mathbf{W}^2_{C2}$ refers to the weight matrix used for the second iteration of classification training ($\mathbf{W}_{C2}$) in the second layer ($\mathbf{W}^2$). Based on the aforementioned conditions, with $j = 2$ fixed, we have flexibility in altering the order of P and C, as well as the number of fine-tuning iterations denoted as i.

Since during our experiments we found out that overfitting was a relevant issue, we decided to use only 2 linear layers, therefore adding a simple Multi-Layer Perceptron (MLP) with 1 hidden layer. This allowed us to benefit from the advantages of not using directly the GNN for the prediction or classification task, which will be described later in Section 3.3, while not increasing too much the complexity of the model. Therefore, the last weight matrix to be applied before prediction will be $\mathbf{W}^2_{T_i}$ (indexing starts from 1).

After the last layer, the outputs will go through a softmax activation function for the classification task (1) and a sigmoid function for the link prediction task (2):

$$\mathbf{P} = \text{softmax}([\mathbf{W}^2_{C_i}]^T[\mathbf{Z}^2]) \tag{1}$$

$$\hat{\mathbf{A}} = \sigma([\mathbf{Z}']^\mathbf{T}[\mathbf{Z}']) \tag{2}$$

where:

$$\mathbf{Z}' = [\mathbf{W}^2_{P_i}]^T[\mathbf{Z}^2]$$

The loss function for both the classification (3) and link prediction tasks (4) will be the cross-entropy loss:

$$\mathcal{L}_C = -\sum_{i \in S}\sum_{j=1}^{c} y_{ij} \ln p_{ij} \tag{3}$$

$$\mathcal{L}_P = -\sum_{(i,j)\in\mathcal{E}} a_{ij} \ln \hat{a}_{ij} + (1 - a_{ij}) \ln (1 - \hat{a}_{ij}) \tag{4}$$

Where S is the set of nodes with labels, c is the number of node classes, $p_{ij} \in \mathbf{P}$, and $\mathcal{E}$ is the training set of edges.

Being the link prediction a self-supervised learning task, we will use as samples the edges that are currently present in the graph along with edges generated randomly by adopting negative sampling. The goal of the network, therefore, is to learn whether the edge exists in the graph or not. To avoid bias and unbalanced classes, we will generate negative samples in a 1:1 ratio with the positive examples.

3.2 Algorithm

Our idea is to merge the training of classification and link prediction, exploiting the implicit encoding function generated by the first training as a starting point for the second one. In our case, we will fine-tune the classification weights by training on link prediction and then go back to the starting task, as described in Figure 1. The aim is to improve the accuracy of classification by, instead of using randomly initialized weights, using knowledge acquired from the link prediction task.

Since the MLP for the several stages is independent from each other and randomly initialized, we must avoid that the learning done on the previous steps gets ruined by a randomly initialized MLP that, of course, is very unlikely to produce a correct result when untrained. To handle this, for the training iterations following the first one, we will train for h epochs, a fraction of the total number of epochs H, only the MLP, and then for the remaining H - h epochs, we will train the whole network, made of the GNN and MLP.

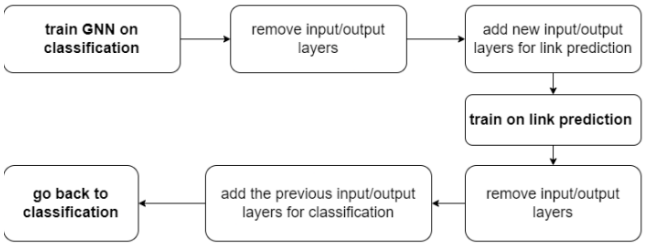

FIGURE 1: flowchart

At this stage, we tried to find the best parameters for an execution flow of the type C, P, C.

3.3 The importance of the MLP

In our approach, incorporating a Multi-Layer Perceptron (MLP) in graph tasks might seem unconventional. Generally, using MLPs in such tasks is discouraged, as they process input features linearly, potentially overlooking edge connections between nodes. In this case, however, the MLP proved crucial for the following reasons:

1. When training for Link Prediction, the network needs to generate embeddings of a specific size. Relying solely on the Graph Neural Network (GNN) would constrain the embedding size to match the number of classes.

   For instance, with a GNN featuring self-loops and shared parameters, the last layer's matrix for classification would be $\mathbf{W} \in \mathbb{R}^{h \times C}$, where C is the number of classes and h is the size of the hidden representation. With a small C (e.g., $C = 3$ for datasets like PubMed), we sacrifice expressive power due to the limited class count. Using an MLP, however, gives us the freedom to choose the embedding size independently and then align the MLP's output with the correct number of classes.

2. Without an MLP, transitioning from one task to another results in an immediate loss of information from the previous task. Our goal was to implement a "fine-tuning"-like approach, preventing the network from instantly discarding all prior knowledge just because the embeddings were more tailored to a different task. Although this exploration is not extensively covered here, it may represent a potential robust initialization technique, differing from the random initialization commonly used in classification tasks. This has been explored later on.

In summary, the inclusion of a Multi-Layer Perceptron (MLP) in our graph tasks proves crucial. It provides flexibility in embedding sizes for Link Prediction and ensures a subtle, fine-tuning-like approach to retain knowledge across diverse tasks. The MLP emerges as a key element in enhancing adaptability and overall network performance.

4 Experiments

4.1 Datasets

In the literature, i.e., [2], several datasets are used as a benchmark to report the accuracy of their models as a measure of comparison with other techniques. Consequently, we decided to adopt a selection of those datasets in our experiment:

- The **CiteSeer** dataset consists of 3312 scientific publications classified into one of six classes, with the citation network consisting of 4732 links. Each publication in the dataset is described by a 0/1-valued word vector indicating the absence/presence of the corresponding word from the dictionary, which consists of 3703 unique words.
  By using the training and test sets provided by Planetoid [6], we will use 120 training nodes.
- The **Cora** dataset consists of 2708 scientific publications classified into one of seven classes, with the citation network consisting of 5429 links. Each publication in the dataset is described by a 0/1-valued word vector indicating the absence/presence of the corresponding

word from the dictionary, which consists of 1433 unique words.
By using the training and test sets provided by Planetoid [6], we will use 140 training nodes.

- The **Pubmed** dataset consists of 19717 scientific publications from the PubMed database, pertaining to diabetes and classified into one of three classes. The citation network consists of 44338 links. Each publication in the dataset is described by a TF/IDF weighted word vector from a dictionary consisting of 500 unique words. By using the training and test sets provided by Planetoid [6], we will use 60 training nodes.

Since we are adopting the PyTorch Geometric Library (PyG) [7] for the implementation of the networks, these datasets are easily accessible and can be used thanks to *torch_geometric.datasets.Planetoid* [6].

For link prediction, before training, the graph is preprocessed according to the needs of the evaluation. We randomly selected 10% of the edges as verification edges and 10% as test edges. After this, we deleted them from the graph for the training stage.

We have implemented an efficient approach to test our hyperparameters (i.e., embedding sizes, hidden channels, dropout) that automates the training with all possible combinations of those parameters.

Every time we train a network with different parameters, we save the results in some JSON files to keep track of the scoring of each set of parameters.

4.2 Architectures

We will compare the results of our approach with 3 GNN architectures:

GCN [8]: GCNs use convolutional operations to extract features from the neighbors of each node and capture relationships between nodes in the graph. They introduced neighborhood aggregation with self-loops. This network is easily implementable thanks to the GCNConv layer available in PyG. We used the improved version with $\tilde{\mathbf{A}} = \mathbf{A}+2\mathbf{I}$.

GATv2 [9]: A Graph Attention Network is a GNN that relies on attention weights to appropriately weigh the neighbors of each node when aggregating their hidden representations. According to what is stated by [9], their new architecture, GATv2, consistently outperformed the original GAT [10] and thus should be preferred. The main difference between the 2 architectures is that in GATv2, the weights $\mathbf{W}$ are applied after the concatenation of the inputs x_i, x_j and the attention weights $\mathbf{W^{att}}$ are applied after the activation function. PyG provides the layers for the convolution in both these architectures, and the difference is reported below.

GATv2Conv:

$$e_{ij} = W^t_{att}\text{LeakyReLU}(\mathbf{W}[x_i||x_j])$$

GATConv:

$$e_{ij} = \text{LeakyReLU}(W^t_{att}[\mathbf{W}x_i||\mathbf{W}x_j])$$

We use 8 attention heads for the hidden layer and only 1 attention head in the final layer, as suggested by the authors.

GraphSAGE [11]: GraphSAGE works by sampling and aggregating information from a node's neighborhood, allowing it to capture the graph's structural properties and features. It keeps track of both the hidden representation of the node u and its neighbors by processing them with different weights and then concatenating the result for each hidden representation.

All these architectures are well suited both for link prediction and classification, making them good candidates for our experiments.

4.3 Metrics

To evaluate our models, we will rely on classification accuracy. Intermediate results on the link prediction task can be evaluated by prediction accuracy (thus, the network must predict 1 if the edge should be present in the graph, 0 otherwise).

4.4 Machine For Experiments

The machine used for the experiments has the following characteristics: i5-11600K CPU, 32GB RAM, Nvidia GTX 2070 Super 8GB GPU, and Ubuntu 22.04 OS.

5 Choice of the hyperparameters

The results reported in Table 1 and Table 2 are computed by running 5 times on the test set each of the best models for each of the four tasks shown. Please note that these results were obtained after selecting the best performing model over the validation set.

5.1 GCN

As mentioned above, GCNs use convolutional operations to extract features from the neighbors of each node and capture relationships between nodes in the graph.
During the development of the code, we accumulate some hyperparameter to tune and in our experiments, we fix some one as suggested by [3]: we keep the GCN dropout at 0.5, and hidden channels at 16. We tried to modify these values but we didn't notice particular difference. After some test, we found that a good range of values for the embedding size is, 16-64. The best value may change, depending on the dataset.
During our experiments, when we tried to integrate link prediction, we discovered that the number of epochs must be at least 50-100 for each part of the sequence: classification 1, link prediction, classification 2.
Integrating MLPs, we identified as hyperparameters also the sizes of the hidden layers of each MLPs, each dropout and the fraction of epochs for link prediction and classification 2, where the net is freezed, and the updating of the weights involves only MLPs. In our case we found that a good range for the hidden sizes is 8-16, for MLP dropouts is 0-0.2. A good value for the fraction of epochs to keep the GNN weights frozen is 0.5.

5.2 GATv2

Despite the great results obtained on several tasks related to graph analysis, in recent years some studies led to the development of new architectures such as GAT and GATv2.

In particular, as mentioned above, the goal of these solutions is to increase the expressivity of a GNN by weighting the neighbors of each node during the message passing. In our experiments, we tried to find a correlation between the number of attention heads of the network and the performances of our classification algorithm, in particular we managed to find a number of attention heads that allows the network to perform well on both link prediction and classification, taking also into account that our goal is to optimize the classification, thus this value should contribute to improve especially in the latter stages of training.

Since in both [9] and in [10] the authors used almost the same hyper parameters, we decided to use those values as starting point and we realized that also for our approach most of those values were the best performing ones.

The fact that we were able to use almost the same configuration allows us also to conclude, at the end of the experiments, that the differences in the performance depend mostly on the differences between our algorithms and the standard classification algorithm used in the original papers rather than on the single hyper parameters values.

For this reason we choose to use a 2-layer model and, with respect to the GCN model, in this case we only have 1 additional hyper parameter to test, which is in fact the number of attention heads used in the first layer of the network, since in the second layer we used a single attention heads as the authors of [9] and [10] did.

The first layer consists of $K = 8$ attention heads computing $F' = 8$ features each followed by an ELU activation layer. The second layer uses a single attention head that computes a number of features equal to the number of classes, followed by a softmax activation. The only exceptions are in the GNN+MLP and in the GNN+LinkPred+MLP models that required 8 attention heads in the second layer while training on Pubmed because of the training set size.

In all the architectures that ends with an MLP, the best values for the size of the only hidden layer and the dropout are always 16 and 0.4, respectively, while the fraction of epochs during which the network was freezed to train just the MLP is 0.4.

The learning rates go from 0.01 to 0.05 while the weight decay is almost always around 0.001.

During the validation phase we observed that the best number of attention heads is 8 for all the different settings, since such number of heads allows the model to capture enough relationships in the data without overfitting, while a lower number of attention heads lead to overfitting and a greater number lead to a model too complex that doesn't perform well as the other one.

5.3 GraphSAGE

GraphSAGE, with respect to the other aformentioned architectures, has a few different parameters that need further discussion.

Generalized Neighborhood Aggregation: GraphSAGE was among the first architectures introducing generalized neighborhood aggregation: mean, pooling and LSTM were the first (even though many others came next). According to what the authors stated [11], pooling and LSTM were the best and similar in terms of accuracy. However, the former was faster and thus it was chosen for the experiment.

Mini-Batch Generation: This architecture introduced also the mini-batch generation where small graphs are generated starting from a node and sampling randomly a fixed number of neighbors. This neighbor sampling represented a challenge for the classification and link prediction task and will thus be described further:

- *Classification*: We employed the neighbor sampler (also available in PyG as NeighborLoader). For each node in the set $v \in V$, a subgraph is generated with nodes u such that $d(u, v) \leq K$. In each hop, S_i neighbors are randomly selected at iteration $k = i$, and for each neighbor sampled, S_{i+1} neighbors will be chosen at iteration $k = i + 1$.
  The authors identified the optimal configuration as $K = 2$, $S_1 = 25$, and $S_2 = 10$, but this was determined using large datasets. Unfortunately, this values are not applicable to Cora and Citeseer, given their reduced number of nodes. This could potentially lead to overfitting since every batch would encompass most of the graph. We found the best results with $S_1 = 5$, $S_2 = 2$ (thus, much smaller than the one stated originally).

- *Link Prediction*: Generating mini-batches in this case is not as straightforward. Instead of sampling nodes and their respective neighbors, there is a need to sample a link. The sampling is done as described before, starting from the two nodes incident in the sampled edge. This process is already implemented in the LinkNeighborSampler of PyG. However, as we were unable to find scientific references for the usage of Link Prediction with GraphSAGE, we needed to determine the appropriate number. In this case, the number of nodes picked is doubled ($S_1 \cdot S_2$ *for each node* incident in the edge sampled randomly at the beginning).
  We did not notice many differences by setting these values differently from the one chosen for classification and thus we decided not to change them (thus, $S_1 = 5$, $S_2 = 2$).

Across Pubmed and Cora datasets, an noteworthy observation emerged: employing a large batch size (≥ 1024) coupled with a high dropout rate (0.70) lead promising outcomes, particularly when training extended beyond 100 epochs. Contrarily, attempts to regularize by reducing batch size (increasing regularization) and lowering dropout (decreasing regularization) resulted in comparatively inferior performance.

For what concerns Multi-Layer Perceptrons (MLPs), keeping them simple with just one hidden layer containing 32 neurons turned out to be quite effective. On the other hand, when dealing with Graph Neural Networks (GNNs), it seemed like the sweet spot for the embedding size was 32, and this held true for all the datasets we explored.

Significantly, Pubmed, characterized by a substantial number of nodes, exhibited improved performance when we in-

Dataset	GNN + MLP			GNN + MLP + Link Prediction		
	GCN	GATv2	GraphSAGE	GCN	GATv2	GraphSAGE
Cora	0.777 ±0.012	**0.793±0.007**	0.766 ±0.009	0.773 ±0.016	**0.793±0.014**	0.778 ±0.005
Citeseer	0.614 ±0.006	0.673±0.002	**0.674 ±0.008**	0.639 ±0.016	0.654±0.017	**0.701 ±0.004**
Pubmed	0.737 ±0.011	0.768±0.005	**0.776 ±0.003**	0.774 ±0.007	0.744±0.019	**0.788 ±0.001**

TABLE 1: Metrics computation over the test sets when using the MLP at the end of each GNN

Dataset	GNN			GNN + Link Prediction		
	GCN	GATv2	GraphSAGE	GCN	GATv2	GraphSAGE
Cora	0.778 ±0.009	**0.791±0.011**	0.774 ±0.011	0.783 ±0.007	**0.796±0.008**	0.780 ±0.007
Citeseer	0.657 ±0.006	**0.699±0.005**	0.668 ±0.006	**0.669±0.012**	0.660±0.017	0.668 ±0.012
Pubmed	0.763 ±0.004	0.767±0.007	**0.778 ±0.007**	0.767 ±0.005	0.769±0.013	**0.778 ±0.010**

TABLE 2: Metrics computation over the test sets by using the GNNs directly without MLP

creased the hidden layer's dimensionality in GraphSAGE. In contrast, for Cora and Citeseer, which are comparatively less complex, opting for smaller sizes in the hidden representation proved to be a successful strategy.

6 Results

The results show distinct behaviors for GCN and GraphSAGE in comparison to GATv2. Tables 1 and 2 highlight the noteworthy impact of incorporating link prediction on the performance of GCN and GraphSAGE.

While the straightforward attachment of an MLP to the GNN, without considering the graph connections, led to a degradation in performance for the majority of cases, an interesting difference occurred when integrating the MLP with the Link Prediction task. Notably, this combined approach not only frequently surpassed the results of the basic GNN but also, in some instances, outperformed the GNN with link prediction pre-training.

This phenomenon can be observed more in GraphSAGE compared to GCN, underscoring the effectiveness of the combined GNN and Link Prediction approach, especially when coupled with an MLP.

On the other hand, the results we obtained by training a GATv2 network on both link prediction and classification were less promising compared to the ones achieved with GCN and GraphSAGE.

In particular, we see that in Cora it achieved great performance with all the different approaches, while in Citeseer not attaching any linear layer was the best approach. Finally, Although the improvement was minimal, Pubmed gained benefit from using MLP combined with GNN and link prediction. In general, the increase in complexity when attaching the MLP was not worth it for this network since the improvement in performance was relatively small.

It is noteworthy to say that in our implementation, even by using the exact same hyperparameters stated by the authors of the architecture, we were not able to achieve their performance with the simple GNN. In our opinion, this may be related to different nodes used in the training sets. Nevertheless, the most relevant aspect is the improvement that was achieved.

7 Conclusions

In our research, we explored the potential benefits of combining node classification and link prediction tasks in graph neural networks (GNNs). Specifically, we investigated whether integrating these two tasks could enhance the overall performance of GNNs, which are commonly used for tasks involving non-Euclidean data representations, such as graphs.

The experimentation involved three popular GNN architectures: Graph Convolutional Network (GCN), Graph Attention Network version 2 (GATv2), and GraphSAGE. We introduced an additional component called a Multi-Layer Perceptron (MLP) to aid in the training process.

Our findings revealed interesting insights. While adding MLP to the main task did not consistently improve performance, the combination of MLP and a link prediction task showed promise, especially in the case of GraphSAGE. The results varied based on the architecture and dataset.

In conclusion, our study highlights the potential advantages of integrating auxiliary tasks like link prediction alongside node classification, providing valuable insights into optimizing graph-based models.

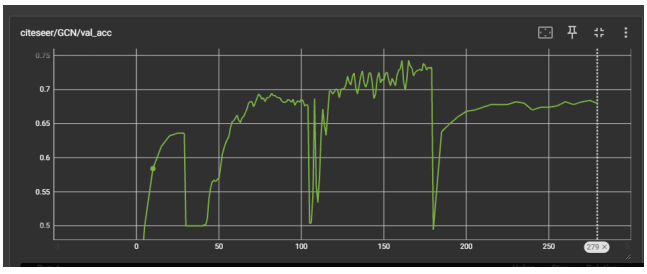

FIGURE 2: the validation accuracy of one of the best run for citeseer and GCN during the tuning: epochs 0-30 = classification 1, 31-110 = link prediction freezed, 111-180 link prediction not freezed, 181-280 = classification freezed. The improving between the end of the classification 1 and the end of the classification 2 in this case was almost 0.1

References

[1] S. A. Fadaee and M. A. Haeri, "Classification using link prediction," CoRR, vol. abs/1810.00717, 2018. [Online]. Available: http://arxiv.org/abs/1810.00717

[2] Z. Wu, M. Zhan, H. Zhang, Q. Luo, and K. Tang, "Mtgcn: A multi-task approach for node classification and link prediction in graph data," Information Processing & Management, vol. 59, no. 3, p. 102902, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0306457322000292

[3] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," arXiv preprint arXiv:1609.02907, 2016.

[4] K. Oono and T. Suzuki, "Graph neural networks exponentially lose expressive power for node classification," arXiv preprint arXiv:1905.10947, 2019.

[5] X. Wu, Z. Chen, W. Wang, and A. Jadbabaie, "A non-asymptotic analysis of oversmoothing in graph neural networks," arXiv preprint arXiv:2212.10701, 2022.

[6] "Pytorch geometric planetoid datasets." [Online]. Available: https://pytorch-geometric.readthedocs.io/en/latest/generated/torch_geometric.datasets.Planetoid.html

[7] "Pytorch geometric." [Online]. Available: https://pyg.org/

[8] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," 2017.

[9] S. Brody, U. Alon, and E. Yahav, "How attentive are graph attention networks?" CoRR, vol. abs/2105.14491, 2021. [Online]. Available: https://arxiv.org/abs/2105.14491

[10] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," 2018.

[11] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," 2018.

A Members' contribution

- **Bedin Manuel:**
  - *Hyperparameter tuning on GATv2.*
  - *General bug corrections.*
  - *get_best_params.py*: Functions to check what are the parameters that obtained the best results in a run using grid search.
  - *std_classification*: Files to run standard classification for results comparison.
  - *Features for parameters and results saving.*
- **Formaggio Alberto:**
  - *model.py*: decision of the models to use in the experiment(GCN, SAGE, GATv2), decision of the number of layers and the parameters needed for each model, implementation of the link predictor model (decoding of the embeddings) and implementation of the models.
  - *engine.py*: implementation of the training loop for classification and link prediction. Implementation of accuracy metrics for classification and link prediction. Implementation of mini batch generation for classification and link prediction needed for GraphSAGE.
  - *Hyperparameter tuning on GraphSAGE.*
- **Zebele Daniele:**
  - *utils.py*: implementation of the grid search approach to test hyperparameters.
  - *main.py*: minor changes
  - *bug corrections*
  - *Features for parameters and results saving.*
  - *Hyperparameter tuning on GCN.*

See the attached excel file for a more in-depth explanation of each member's contribution.

The final contribution in percentage, considering the time spent and the realization of the project, is approximately:

- Bedin Manuel: 30%
- Formaggio Alberto: 50%
- Zebele Daniele: 20%