POLITECNICO
MILANO 1863

Advanced Statistical Learning for Complex Data: Network Analysis

20th September 2024

Student: Usevalad Milasheuski

Professor: Francesca Ieva

PhD DADS

Outline

- Motivation
- Problem Statement
- Exploratory Data Analysis (EDA)
- Data Preprocessing
- GNN Models
- Explainability (XAI)
- Results

PhD DADS

Outline

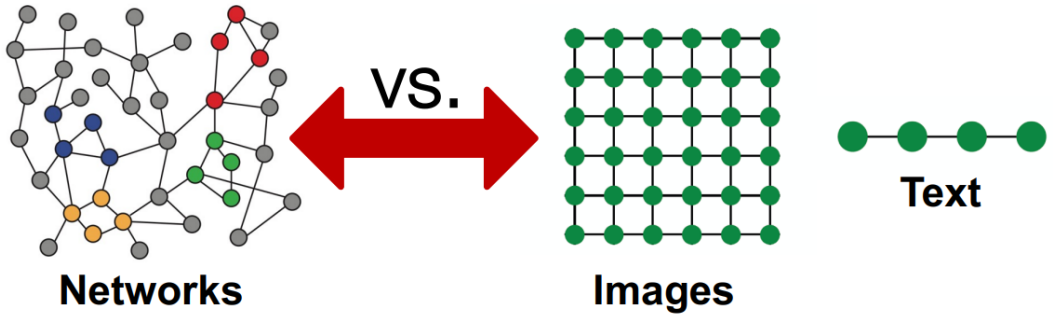
- Motivation
- Problem Statement
- Exploratory Data Analysis (EDA)
- Data Preprocessing
- GNN Models
- Explainability (XAI)
- Results

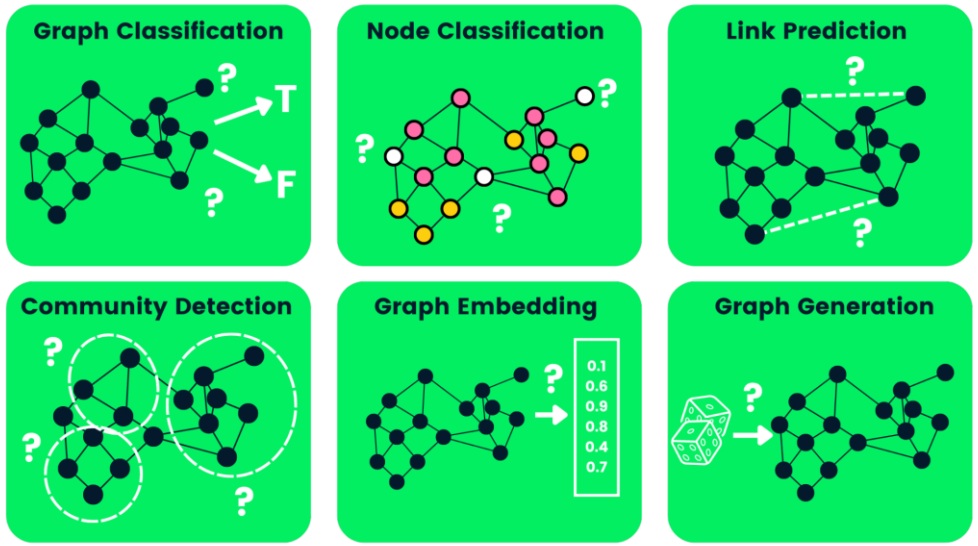
PhD DADS

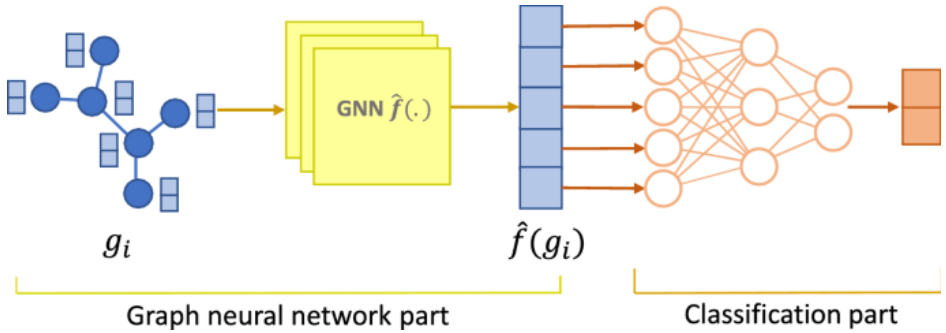
Motivation: Graph Data and GNN

POLITECNICO MILANO 1863

Networks/Graphs are a general language to represent data that describes complex systems. They are useful as they *can represent components* (people, genes, geographical locations, etc.) *and their relationships*. Modern deep learning toolbox is designed for simple sequences & grids. Arbitrary size and complex topological structure (i.e., no spatial locality like grids). No fixed node ordering or reference point, often dynamic and have multimodal feature







PhD DADS

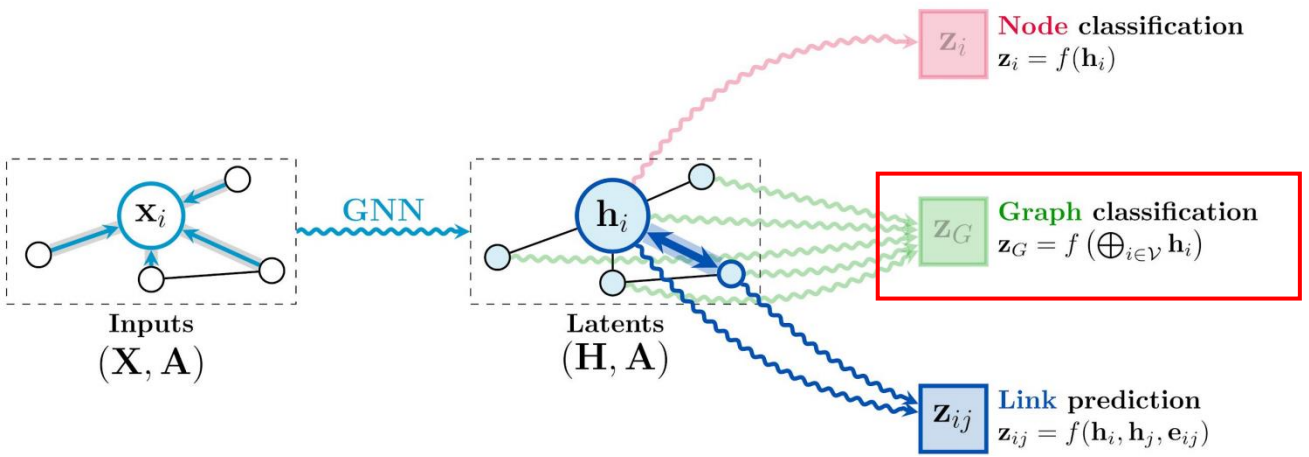
Problem Statement: Graph Classification

Let ***G*** be a set of *K* undirected graphs, where each graph $\mathbf{g}^{(k)}$ has *N* nodes and an arbitrary number of edges. Node-to-node connectivity in $\mathbf{g}^{(k)}$ is given by the weighted adjacency matrix $\boldsymbol{A}^{(k)} \in R^{NxN}$. Each graph is defined on nodes by $\boldsymbol{X}^{(k)} \in R^{NxF}$, with *F* the length of feature vectors. Each graph is associated with a binary label $Y^{(k)} \in \{0, 1\}$. The goal is to develop function ***f(.)***, which would map each graph to the corresponding label

$$\boldsymbol{f}(\mathbf{g}^{(k)}) = f(\boldsymbol{A}^{(k)}, \boldsymbol{X}^{(k)}) \rightarrow Y^{(k)}$$

permutation-invariant

$$f(\mathbf{PX}, \mathbf{PAP}^\top) = f(\mathbf{X}, \mathbf{A})$$



Data:

two synthetic datasets (**balanced** and **unbalanced**), simulated using a *Gaussian graph model* (*K* = 500, *N* = 20, *F* = 1)

	pat_id	conf	class	Prot1	Prot2	Prot3	Prot4	Prot5	Prot6	Prot7	...
0	1	0.165192	0	-0.288741	1.320418	1.246890	-0.587912	0.425350	1.433823	1.573993	...
1	2	-0.513246	0	-1.300974	-1.309550	0.173989	-0.042006	-0.282135	3.165004	0.228712	...
2	3	-0.374802	1	-1.825425	1.211511	2.546314	0.161956	0.211815	4.396854	3.311472	...
3	4	0.367808	1	0.449866	-0.493820	0.975631	1.253092	-1.101147	0.124996	1.165723	...
4	5	0.364785	0	-0.906779	-0.535000	1.317661	0.502248	2.241087	1.893090	0.866129	...

```
Number of graphs: 500
Number of nodes: 20
Number of features per node: 1
All connected: True
Labels: {0: 269, 1: 231}
```

54/46 %

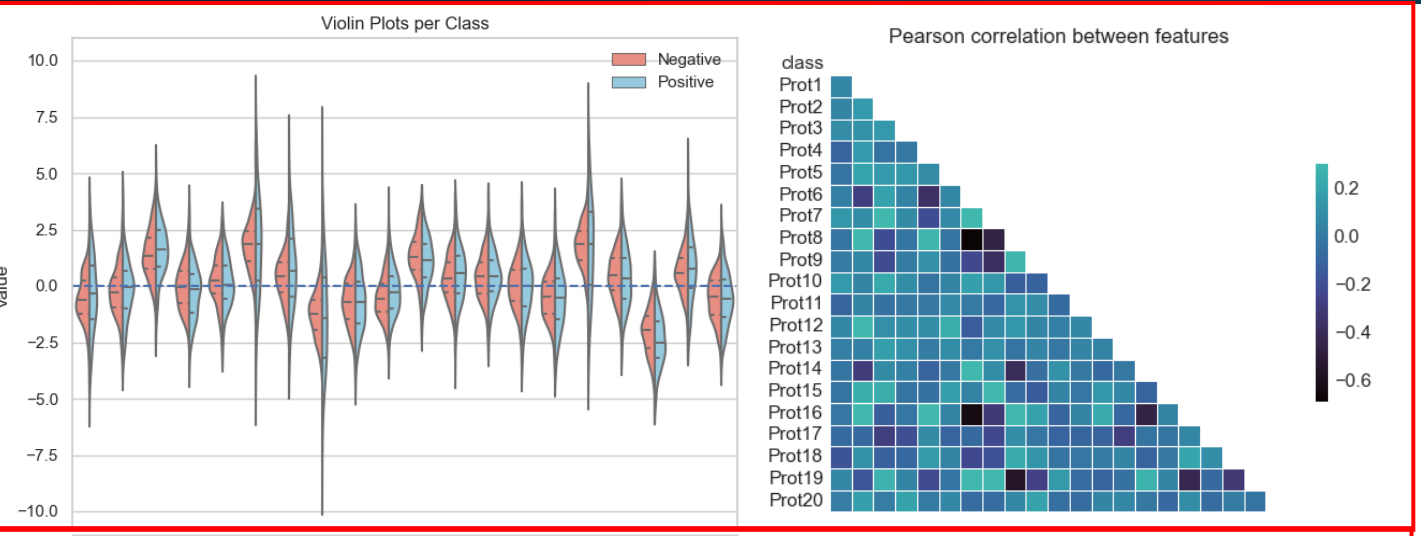
```
Number of graphs: 500
Number of nodes: 20
Number of features per node: 1
All connected: True
Labels: {0: 375, 1: 125}
```

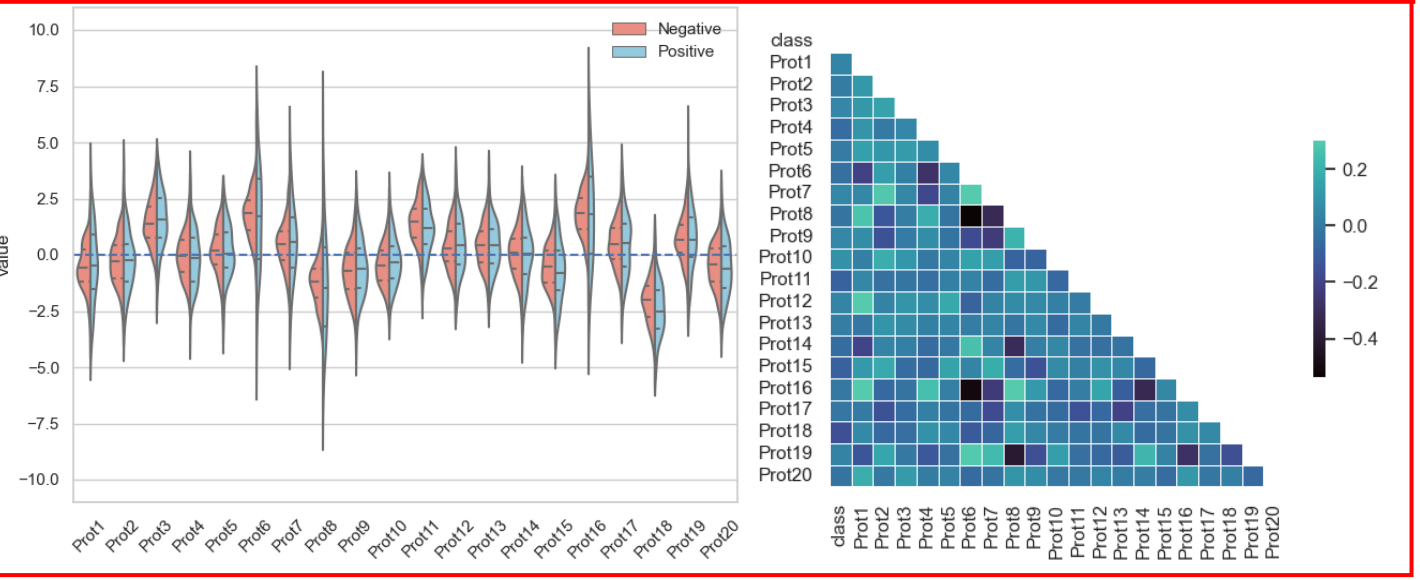
71/28 %

Outline

- Motivation
- Problem Statement
- Exploratory Data Analysis (EDA)
- Data Preprocessing
- GNN Models
- Explainability (XAI)
- Results

Exploratory Data Analysis: Nodes (1)

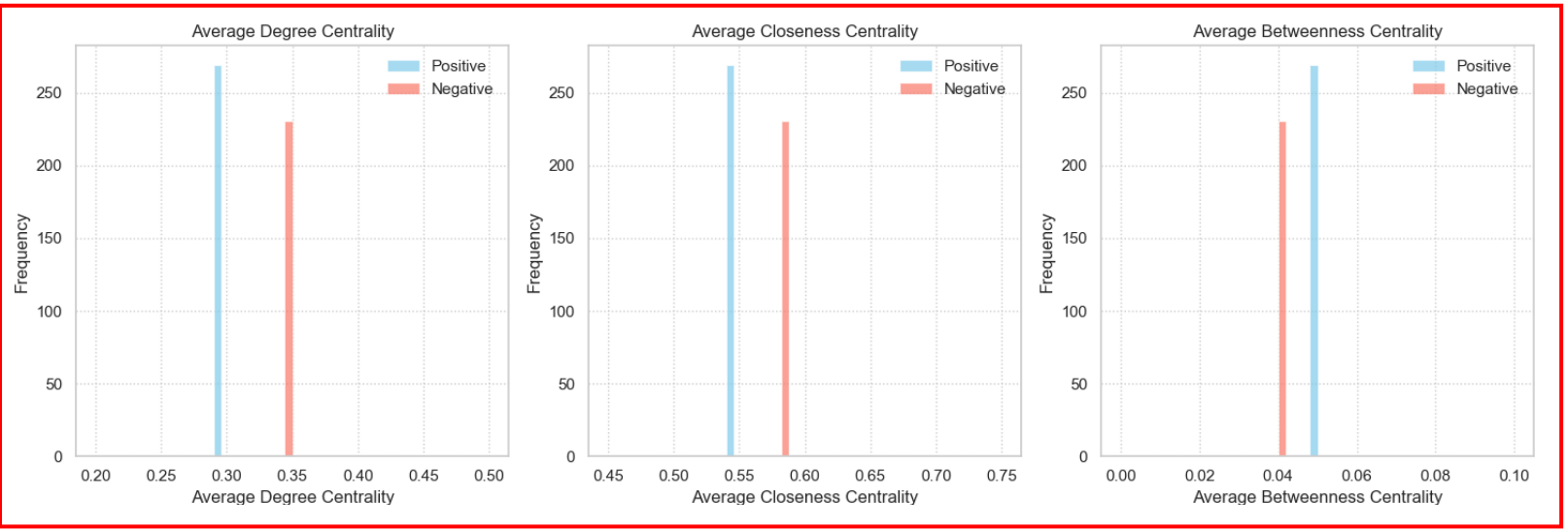




Balanced Dataset

Unbalanced Dataset

Exploratory Data Analysis: Nodes (2)

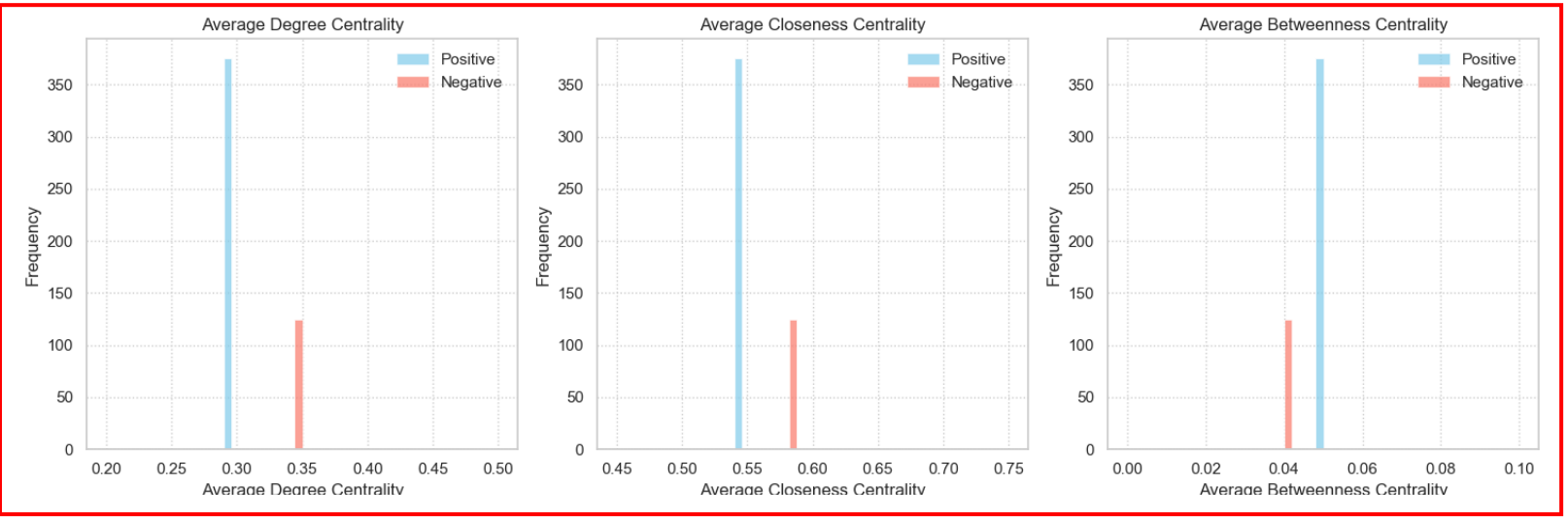


Balanced Dataset

$$C_D(i) = \frac{d(i)}{n-1}.$$

$$C(u) = \frac{n-1}{\sum_{v=1}^{n-1} d(v,u)}$$

$$c_B(v) = \sum_{s,t \in V} \frac{\sigma(s,t|v)}{\sigma(s,t)}$$



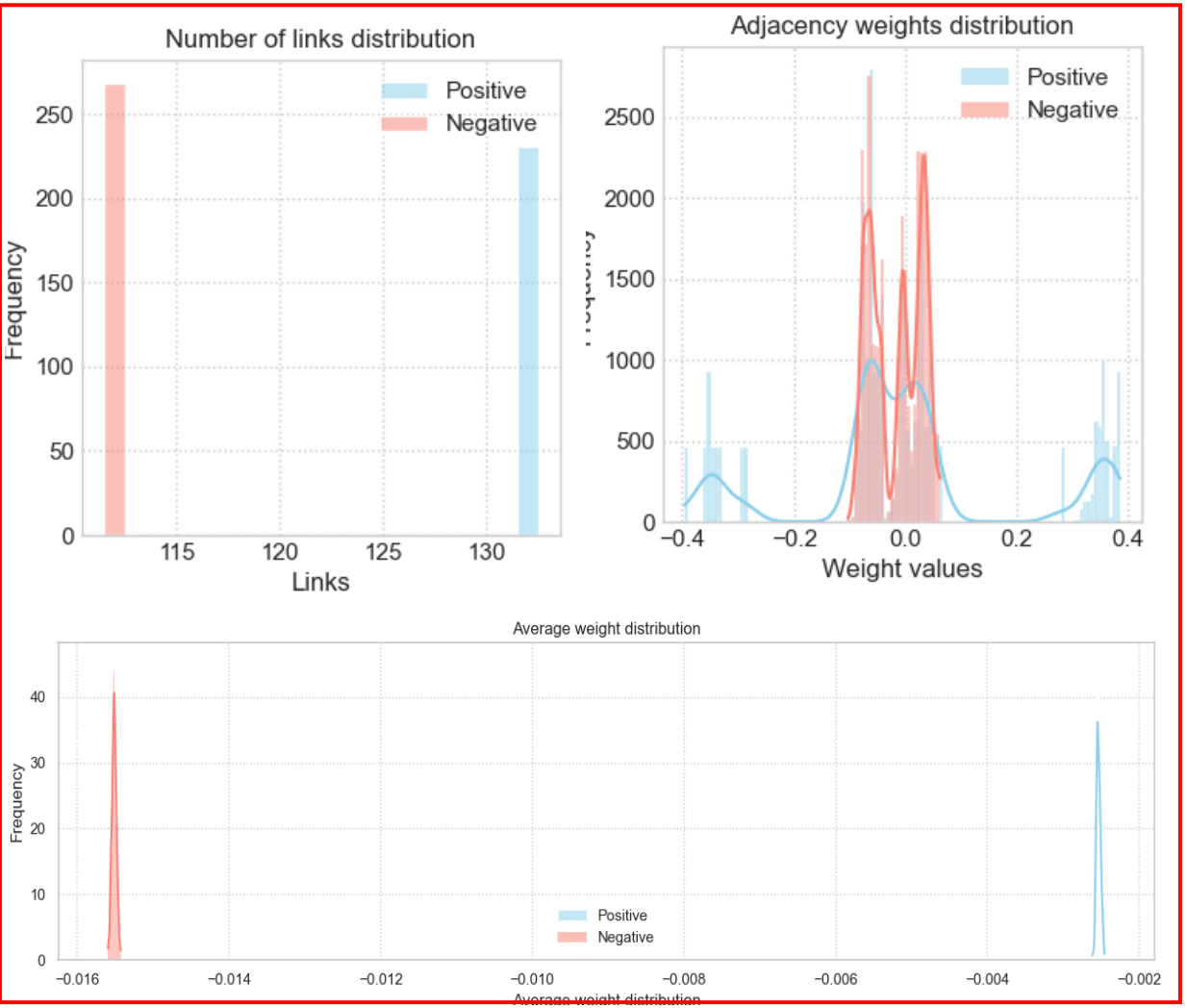
Unbalanced Dataset

PhD DADS

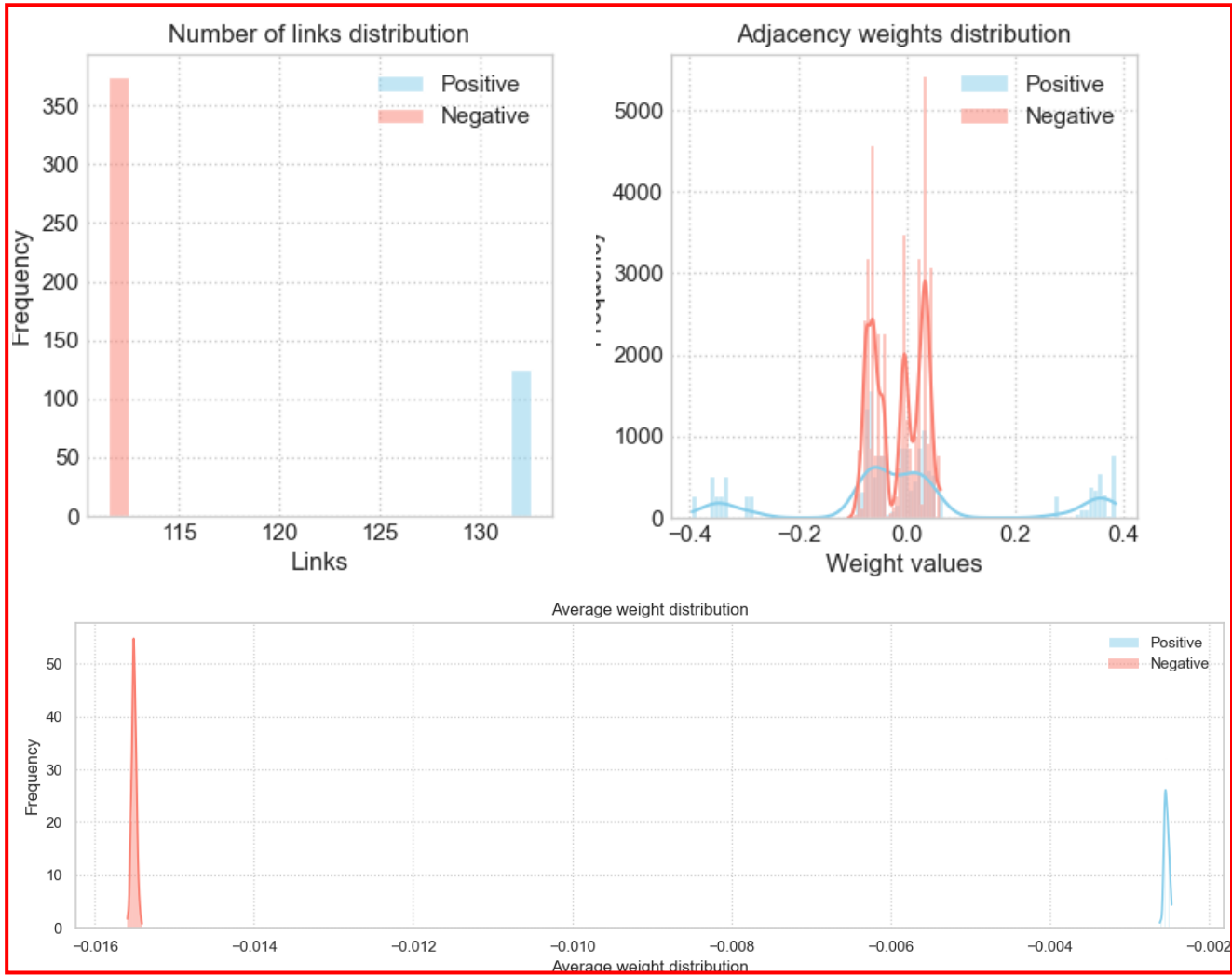
Exploratory Data Analysis: Links

POLITECNICO MILANO 1863

Balanced Dataset



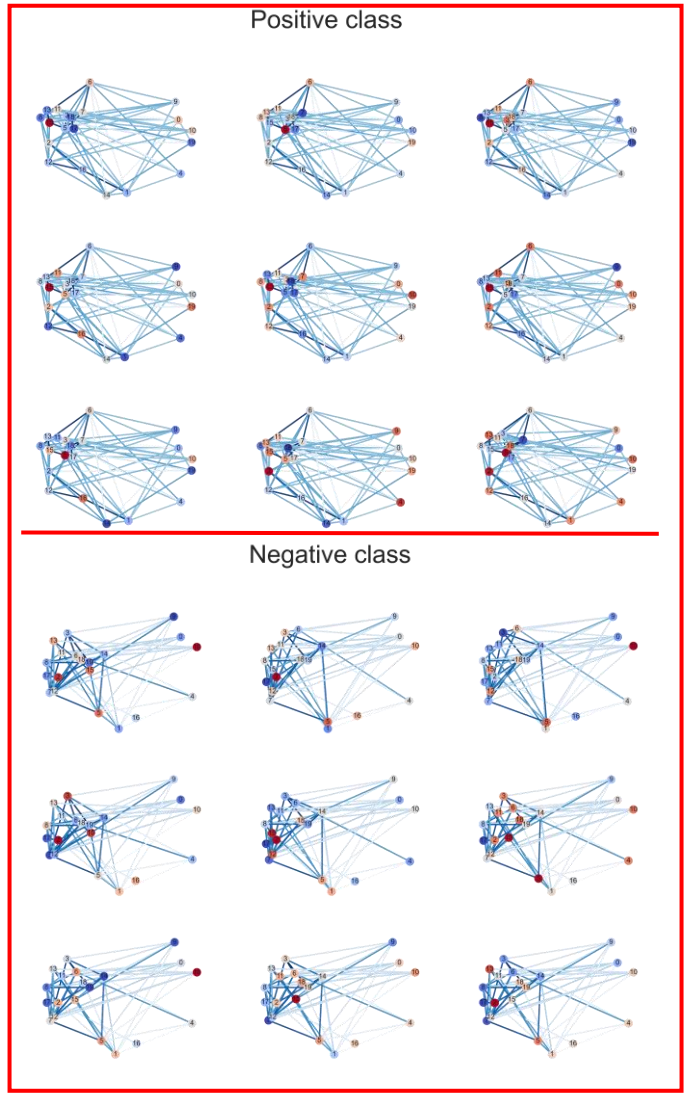
Unbalanced Dataset

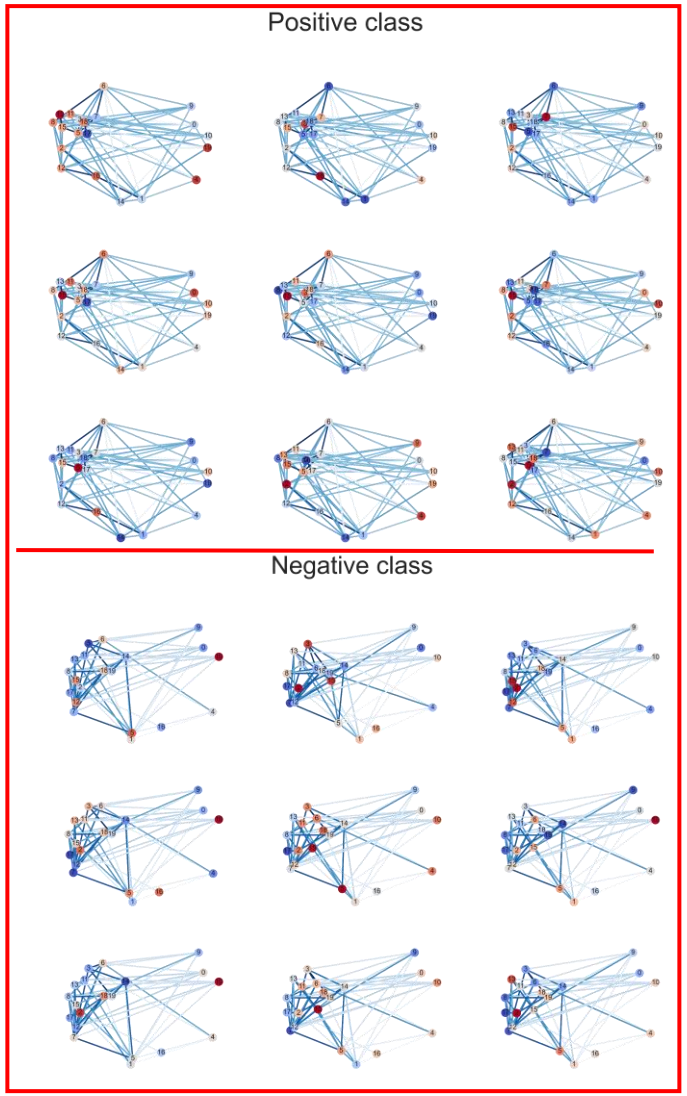


Exploratory Data Analysis: Graphs

POLITECNICO MILANO 1863

Balanced Dataset





Unbalanced Dataset

PhD DADS

Exploratory Data Analysis

Observations:

- Based on geometry, label and node-related distributions, both datasets were generated from a Gaussian graph model with the same parameters.
- Some average statistic over the graphs allows us clearly differentiate between classes.
- The feature per node seem have small discrepancy between each other, as well as small correlation to the target.

Remarks:

- Negative values in adjacency matrices may cause problems (unless treated as features) -> *feature preprocessing*
- The feature matrix $\boldsymbol{X}^{(k)} \in R^{NxF}$, where $F = 1$ might not be sufficient enough for the final prediction -> *feature engineering*

Outline

- Motivation
- Problem Statement
- Exploratory Data Analysis (EDA)
- Data Preprocessing
- GNN Models
- Explainability (XAI)
- Results

PhD DADS

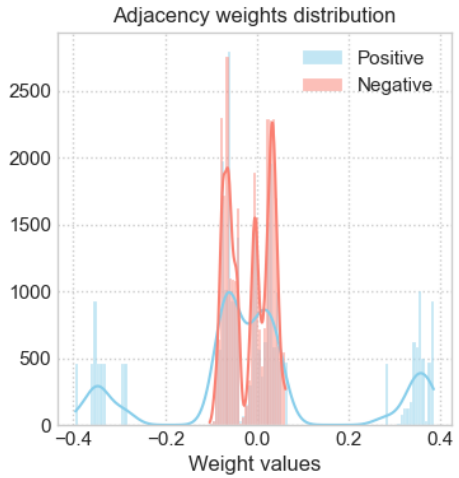
Data Preprocessing: Standardization & Scaling

$$x_i \sim \mathcal{N}(\mu_i, \sigma_i)$$

$$z_i = \frac{x_i - \mu}{\sigma}$$

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

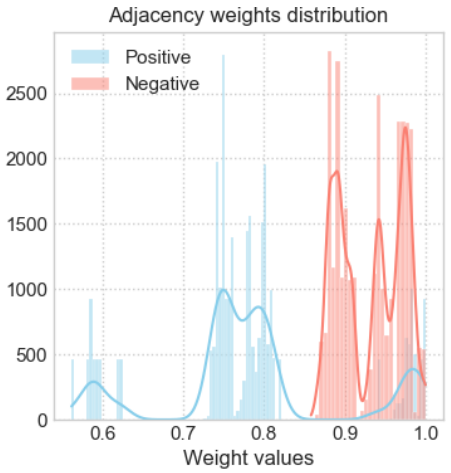
Standardization: ensure each feature contributes equally to the analysis



$$\mathbf{a}_k \in [a_k^{min}, a_k^{max}], a_k^{min} < 0$$

$$b_k^m = \frac{a_k^m - a_k^{min} + 1}{a_k^{max} - a_k^{min} + 1}$$

To non-zero values only!



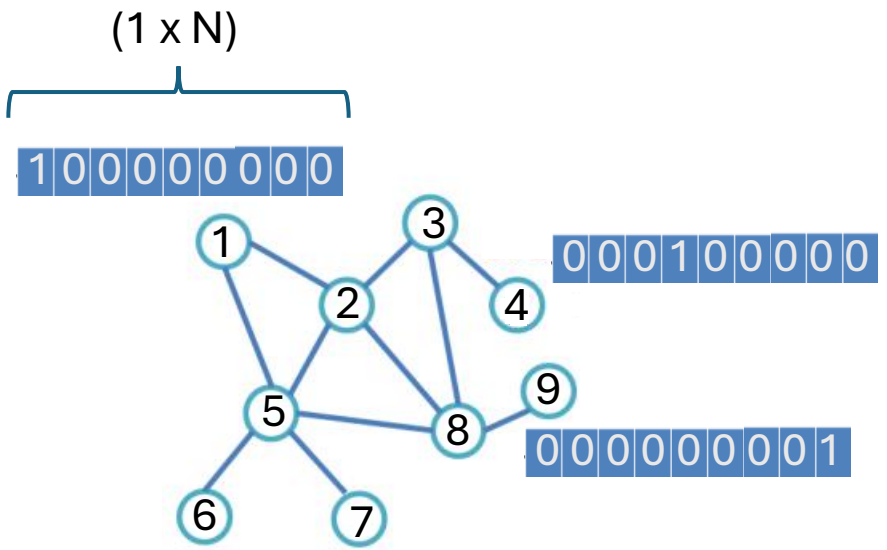
$$\mathbf{b}_k = [\frac{1}{a_k^{max} - a_k^{min} + 1}, 1]$$

Scaling: maintain the relationships among points while falling within a defined range

Data Preprocessing: Position Encoding

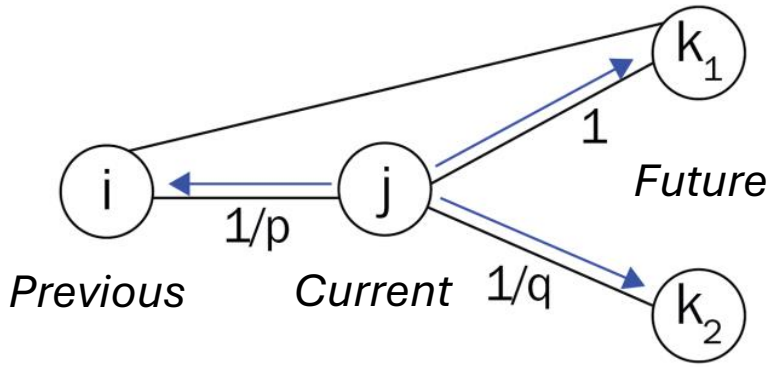
Node features Note that unlike the shallow embedding methods discussed in Part I of this book, the GNN framework requires that we have node features $\mathbf{x}_u, \forall u \in \mathcal{V}$ as input to the model. In many graphs, we will have rich node features to use (e.g., gene expression features in biological networks or text features in social networks). However, in cases where no node features are available, there are still several options. One option is to use node statistics—such as those introduced in Section 2.1—to define features. Another popular approach is to use *identity features*, where we associate each node with a one-hot indicator feature, which uniquely identifies that node.[a]

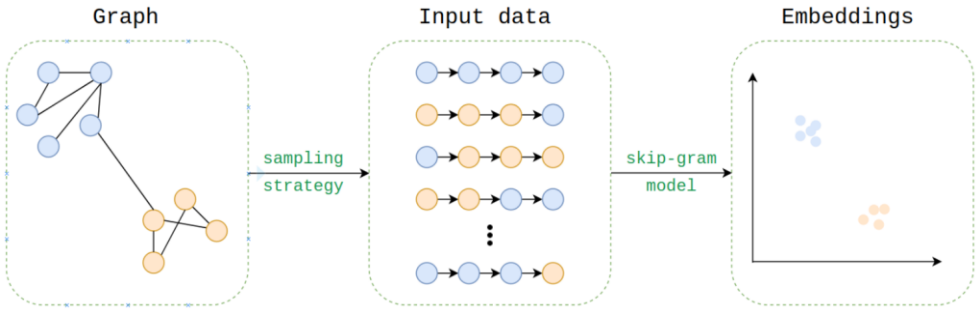
[a]Note, however, that the using identity features makes the model transductive and incapable of generalizing to unseen nodes.

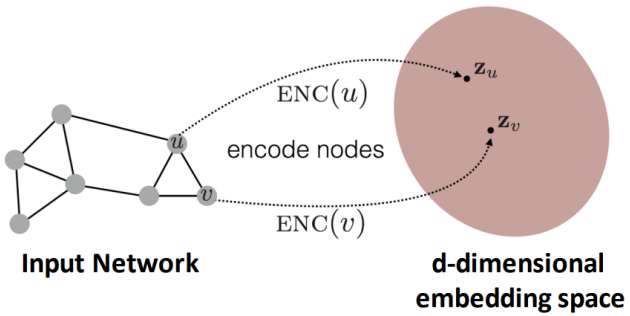


Source: Hamilton, William L., Graph Representation Learning

Data Preprocessing: Node2Vec







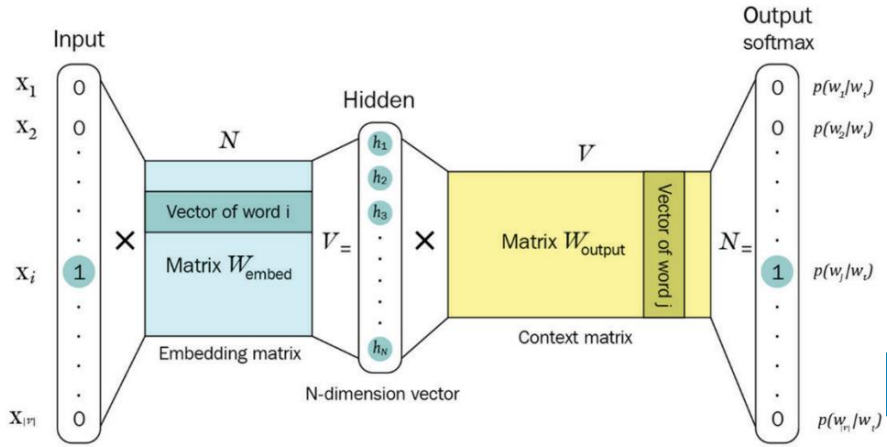
$\text{similarity}(u, v) \approx \boldsymbol{z}_v^T \boldsymbol{z}_u$

- $\pi_{jk} = \alpha(i,k)\,\omega_{jk}$, $\alpha(i,k)$ – search bias, ω_{jk} - edge weight.

$$\alpha(a,b) = \begin{cases} \frac{1}{p} \; if \; d_{ab} = 0 \\ 1 \; if \; d_{ab} = 1 \\ \frac{1}{q} \; if \; d_{ab} = 2 \end{cases}$$

Use flexible, biased random walks that can trade off between local and global views of the network

- Return parameter $\boldsymbol{p}$: Return back to the previous node (increase $\boldsymbol{p}$ -> new nodes (DFS)).
- In-out parameter $\boldsymbol{q}$: Moving outwards (increase $\boldsymbol{q}$ -> closer nodes (BFS))

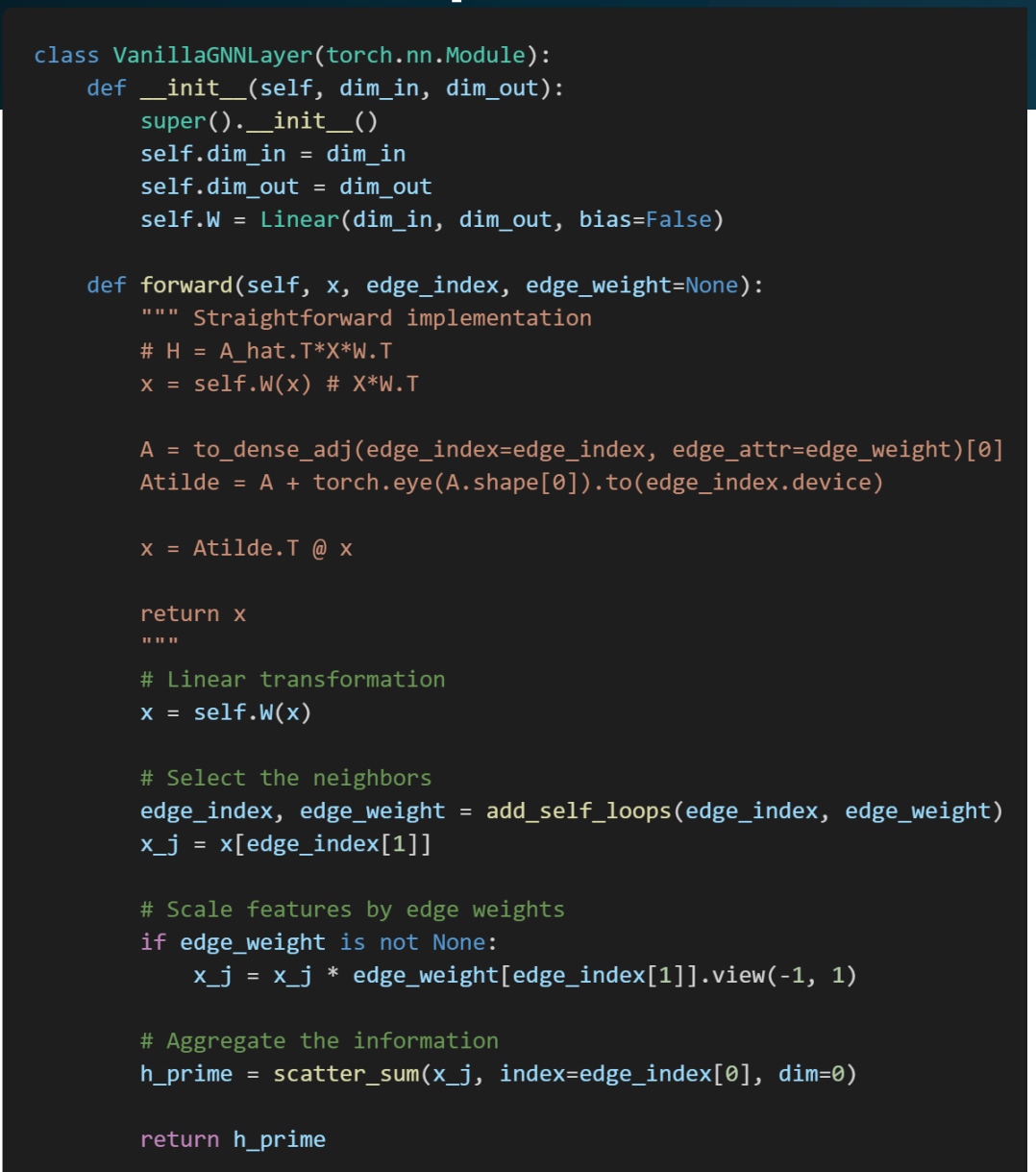


Outline

- Motivation
- Problem Statement
- Exploratory Data Analysis (EDA)
- Data Preprocessing
- GNN Models
- Explainability (XAI)
- Results

PhD DADS

Graph Neural Networks: Vanilla GNN



```python
class VanillaGNNLayer(torch.nn.Module):
    def __init__(self, dim_in, dim_out):
        super().__init__()
        self.dim_in = dim_in
        self.dim_out = dim_out
        self.W = Linear(dim_in, dim_out, bias=False)

    def forward(self, x, edge_index, edge_weight=None):
        """ Straightforward implementation
        # H = A_hat.T*X*W.T
        x = self.W(x) # X*W.T

        A = to_dense_adj(edge_index=edge_index, edge_attr=edge_weight)[0]
        Atilde = A + torch.eye(A.shape[0]).to(edge_index.device)

        x = Atilde.T @ x

        return x
        """
        # Linear transformation
        x = self.W(x)

        # Select the neighbors
        edge_index, edge_weight = add_self_loops(edge_index, edge_weight)
        x_j = x[edge_index[1]]

        # Scale features by edge weights
        if edge_weight is not None:
            x_j = x_j * edge_weight[edge_index[1]].view(-1, 1)

        # Aggregate the information
        h_prime = scatter_sum(x_j, index=edge_index[0], dim=0)

        return h_prime
```

Basic implementation of a graph neural network layer with PyTorch. The information is aggregated by looking at the node's neighborhood constrained by the adjacency matrix.

$$h_A = \sum_{i \in \mathcal{N}_A} x_i W^T$$

$$H = \tilde{A}^T X W^T$$

$$\tilde{A} = A + I$$

Graph Neural Networks: Vanilla GCN

POLITECNICO MILANO 1863

```python
###################### VanillaGCN ####################
class VanillaGNNLayer(torch.nn.Module):
    def __init__(self, dim_in, dim_out):
        super().__init__()
        self.dim_in = dim_in
        self.dim_out = dim_out
        self.W = Linear(dim_in, dim_out, bias=False)

    def forward(self, x, edge_index, edge_weight=None):
        """ Straightforward implementation
        # H = D_hat^-1*A_hat.T*D_hat^-1*X*W.T

        x = self.W(x) # X*W.T

        A = to_dense_adj(edge_index=edge_index, edge_attr=edge_weight)[0]
        Atilde = A + torch.eye(A.shape[0]).to(edge_index.device)

        Dtilde = torch.diag(1 / torch.sqrt(torch.sum(Atilde, dim=0))).to(edge_index.device)
        adj_norm = Dtilde @ Atilde.T @ Dtilde

        x = adj_norm @ x

        return x
        """
        # Linear transformation
        x = self.W(x)

        # Select the neighbors
        edge_index, edge_weight = add_self_loops(edge_index, edge_weight)
        x_j = x[edge_index[1]]

        # Scale features by edge weights
        if edge_weight is not None:
            x_j = x_j * edge_weight[edge_index[1]].view(-1, 1)

        # Compute degree d
        d = scatter_sum(torch.ones_like(edge_index[0]), index=edge_index[0], dim=0)

        # Compute convolution weights
        d_i = torch.index_select(d, index=edge_index[0], dim=0)
        d_j = torch.index_select(d, index=edge_index[1], dim=0)

        # Avoid division by zero
        d_i[d_i == 0] = 1e-10
        d_j[d_j == 0] = 1e-10

        w = torch.unsqueeze(1 / torch.sqrt(d_i * d_j), -1)

        # Aggregate the information
        h_prime = scatter_sum(x_j*w, index=edge_index[0], dim=0)

        return h_prime
```

Aggregate the information by taking into account this difference in node degree to avoid large values for nodes with large number of neighbors. And assigning higher weights to nodes with few neighbors.

$$h_i = \sum_{j \in \mathcal{N}_i} \frac{1}{\sqrt{\deg(i)}\sqrt{\deg(j)}} x_j W^T$$

$$H = \tilde{D}^{-\frac{1}{2}} \tilde{A}^T \tilde{D}^{-\frac{1}{2}} X W^T$$

PhD DADS

Graph Neural Networks: Vanilla GAT

POLITECNICO MILANO 1863

```python
class GATHead(torch.nn.Module):
    def __init__(self, in_features, out_features):
        super().__init__()
        self.W = Linear(in_features, out_features, bias=False)
        self.a = Linear(2*out_features, 1, bias=False)
        self.leakyrelu = LeakyReLU(0.2)


    def forward(self, x, edge_index, edge_weight=None):
        """ Straightforward implementation
        # H = A_hat.T*W_alpha*X*W.T

        x = self.W(x) # X*W.T

        A = to_dense_adj(edge_index=edge_index, edge_attr=edge_weight)[0]
        Atilde = A + torch.eye(A.shape[0]).to(edge_index.device)


        a_input = torch.cat([x[edge_index[0]], x[edge_index[1]]], dim=1)
        e = self.leakyrelu(self.a(a_input))

        E = torch.zeros_like(A.shape)
        E[edge_index[0], edge_index[1]] = e[0]

        W_alpha = F.softmax(E, dim=1)

        x = A.T @ W_alpha @ x

        return x
        """
        # Linear transformation
        h = self.W(x)

        # Compute attention coefficients
        x_i, x_j = h[edge_index[0]], h[edge_index[1]]
        a_input = torch.cat( [x_i, x_j], dim=1)
        e = self.leakyrelu(self.a(a_input))

        # Compute attention weights using softmax
        attention = softmax(e, edge_index[0])

        # Apply attention weights and ggregate the information
        h_prime = scatter_sum(attention * x_j, index=edge_index[0], dim=0)

        return h_prime
```

```python
###################### VanillaGAT ####################
class VanillaGATLayer(torch.nn.Module):
    def __init__(self, dim_in, dim_out, heads, concat=False):
        super().__init__()
        self.heads = heads
        self.dim_in = dim_in
        self.dim_out = dim_out
        self.concat = concat

        self.attention_heads = ModuleList([
            GATHead(dim_in, dim_out) for _ in range(heads)
        ])

    def forward(self, x, edge_index, edge_weight=None):
        # Apply each attention head
        head_outputs = [att_head(x, edge_index) for att_head in self.attention_heads]

        if self.concat:
            # Concatenate all head outputs along the features
            return torch.cat(head_outputs, dim=1)
        else:
            # Average the head outputs along the heads
            return torch.mean(torch.stack(head_outputs), dim=0)
```

$$h_i = \frac{1}{n}\sum_{k=1}^{n} h_i^k = \frac{1}{n}\sum_{k=1}^{n}\sum_{j\in\mathcal{N}_i} \alpha_{ij}^k \mathbf{W}^k x_j$$

$$\alpha_{ij} = \frac{exp\left(W_{att}^t\ LeakyReLU(\mathbf{W}[x_i \,||\, x_j])\right)}{\sum_{k\in\mathcal{N}_i} exp(W_{att}^t\ LeakyReLU(\mathbf{W}[x_i \,||\, x_k]))}$$

In contrast to the previous layers, the goal of the graph attention layer is to produce weighting factors that also consider the importance of node features by a process called self-attention.

$$H = \tilde{A}^T W_{\text{alpha}} X W^T$$

PhD DADS

Graph Neural Networks: Other Graph Layers

Generalization of GNNs based on the k-WL. The model is strictly stronger then GNNs in terms of distinguishing non-isomorphic (sub-)graphs.

- GraphConv $$h_i = \mathbf{W_1} x_i + \mathbf{W_2} \sum_{j \in \mathcal{N}_i} e_{i,j} x_i$$

- GIN $$h_i' = \phi\left(h_i, f(\{h_j : j \in \mathcal{N}_i\})\right) \qquad h_i' = MLP\left((1+\varepsilon)\cdot h_i + \sum_{j \in \mathcal{N}_i} h_j\right)$$

  - **Aggregate**: This function, f, selects the neighboring nodes that the GNN considers
  - **Combine**: This function, ϕ, combines the embeddings from the selected nodes to produce the new embedding of the target node

The graph isomorphism operator. It is designed for graph classification problem to increase the GNN expressivity *(see k-WL test and injective functions).*

Weisfeiler and Leman Go Neural: Higher-order Graph Neural Networks
How Powerful are Graph Neural Networks?

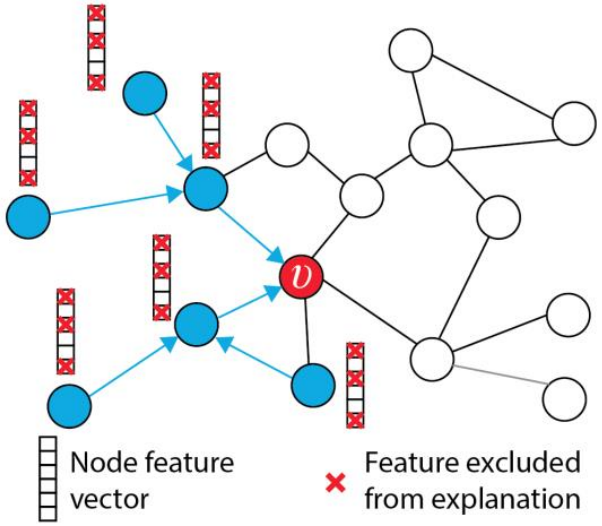
PhD DADS

Outline

- Motivation
- Problem Statement
- Exploratory Data Analysis (EDA)
- Data Preprocessing
- GNN Models
- **Explainability (XAI)**
- Results

PhD DADS

Explainable AI (XAI): Graph Models



- **Perturbation-based methods** mask or modify input features to measure changes in the output.
- *GNNExplainer* implements an edge mask and a feature mask. If the object is important, removing it should drastically change the prediction. On the other hand, if the prediction does not change, it means that this information is redundant or simply irrelevant.

- **Gradient-based methods** analyze gradients of the output to estimate attribution
- *Integrated gradients* technique aims to assign an attribution score to every node and edge. To this end, it uses gradients with respect to the model's inputs. It computes the gradients at all points along the path between and accumulates them.

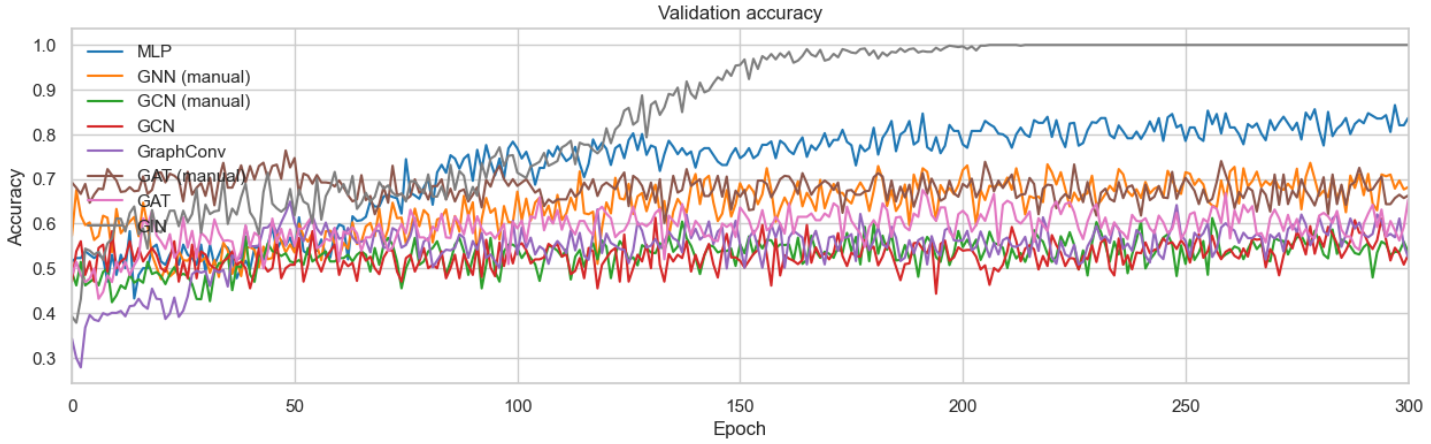
$$IntegratedGrads_i(x) ::= (x_i - x_i') \times \int_{\alpha=0}^{1} \frac{\partial F\left(x' + \alpha \times (x - x')\right)}{\partial x_i} d\alpha$$

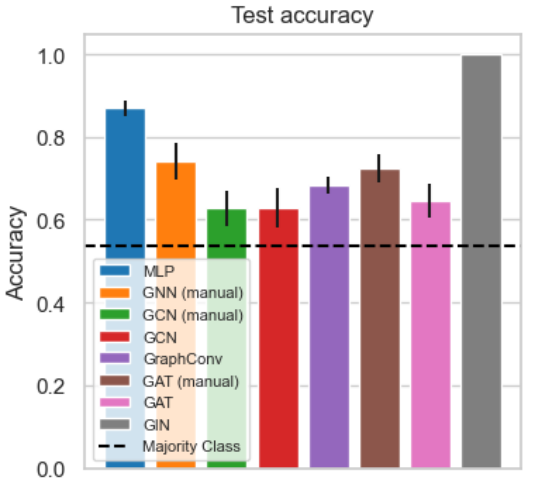
Outline

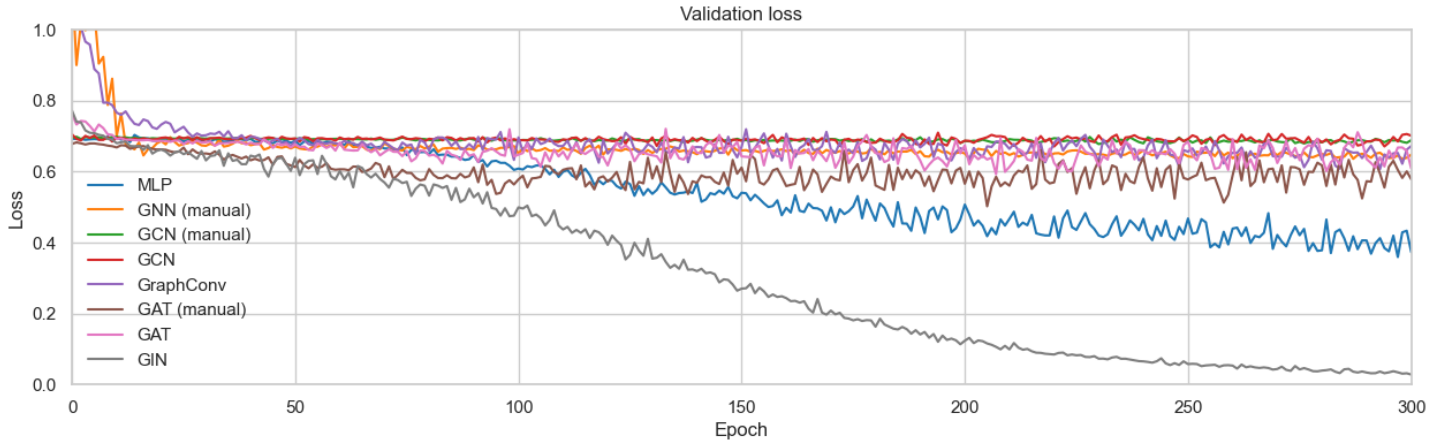
- Motivation
- Problem Statement
- Exploratory Data Analysis (EDA)
- Data Preprocessing
- GNN Models
- Explainability (XAI)
- Results

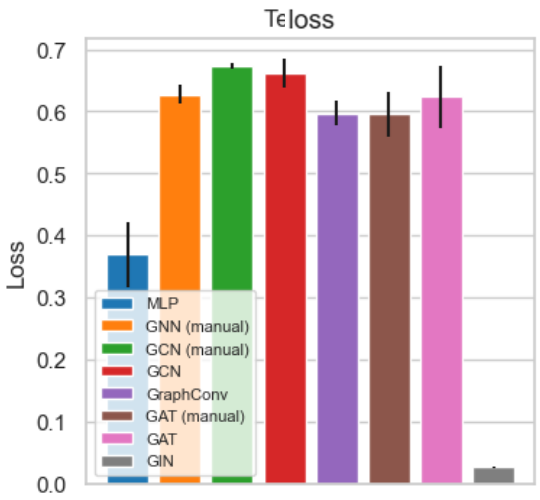
PhD DADS

Results (balanced)

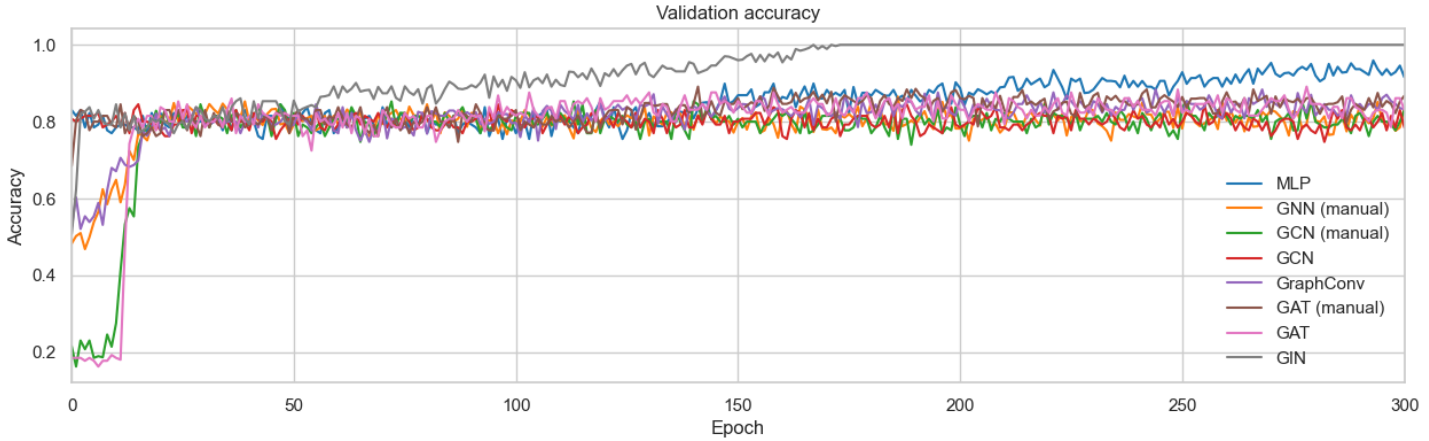


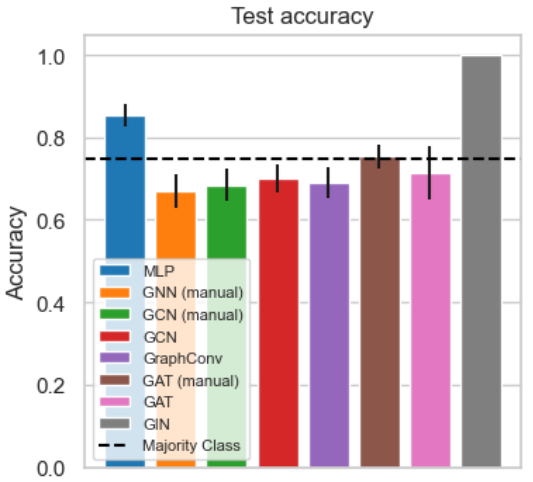


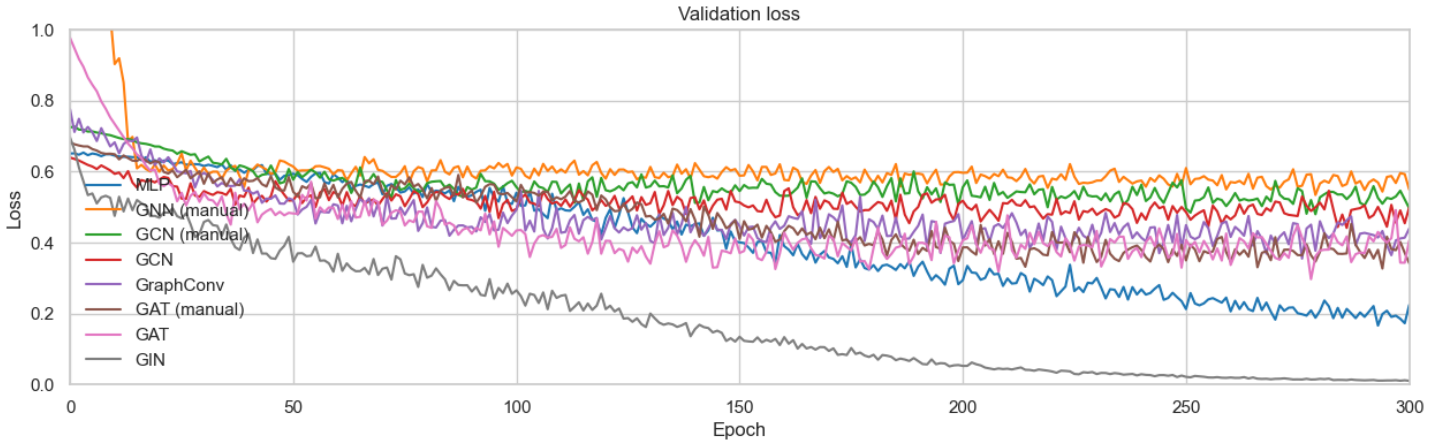


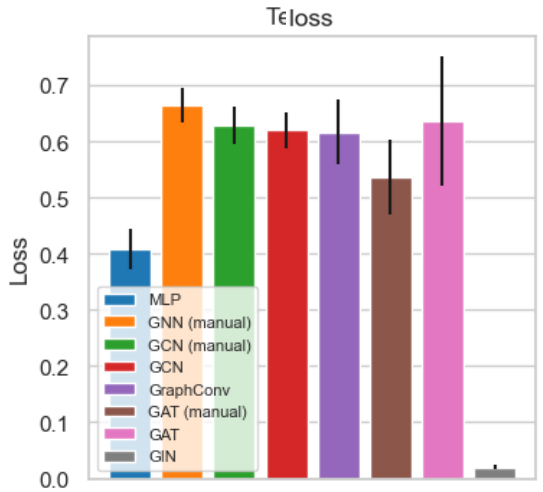


Results (unbalanced)

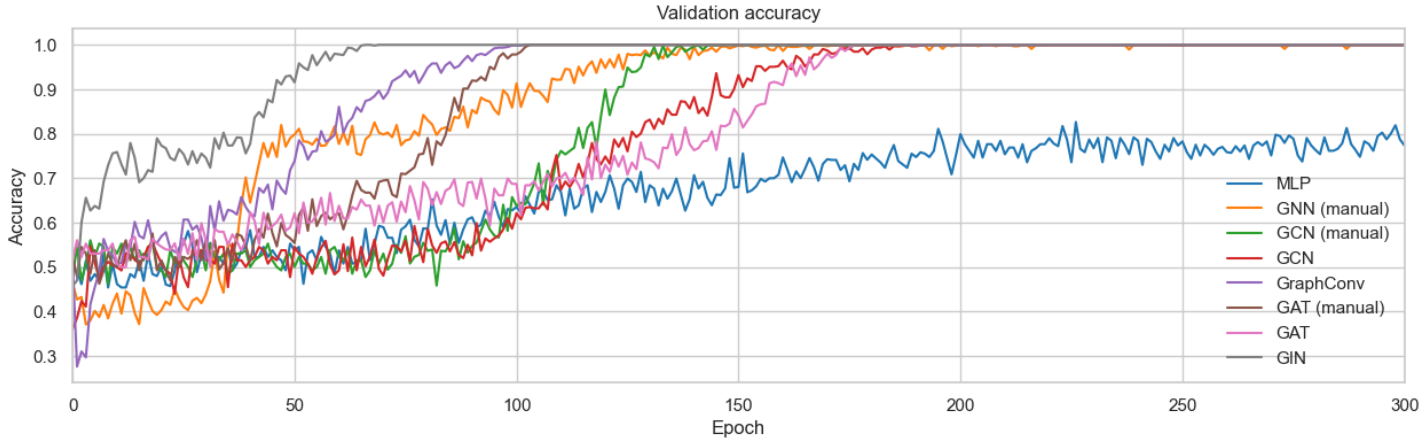


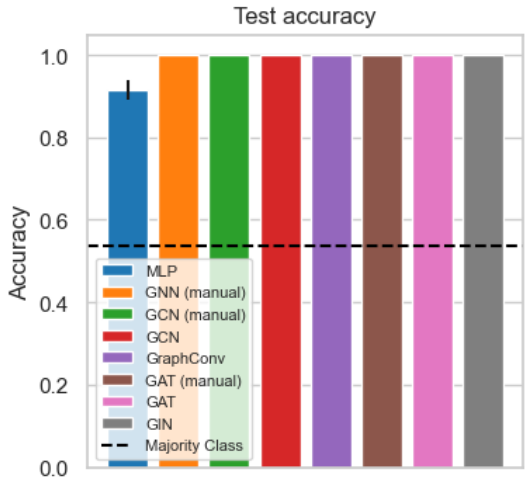


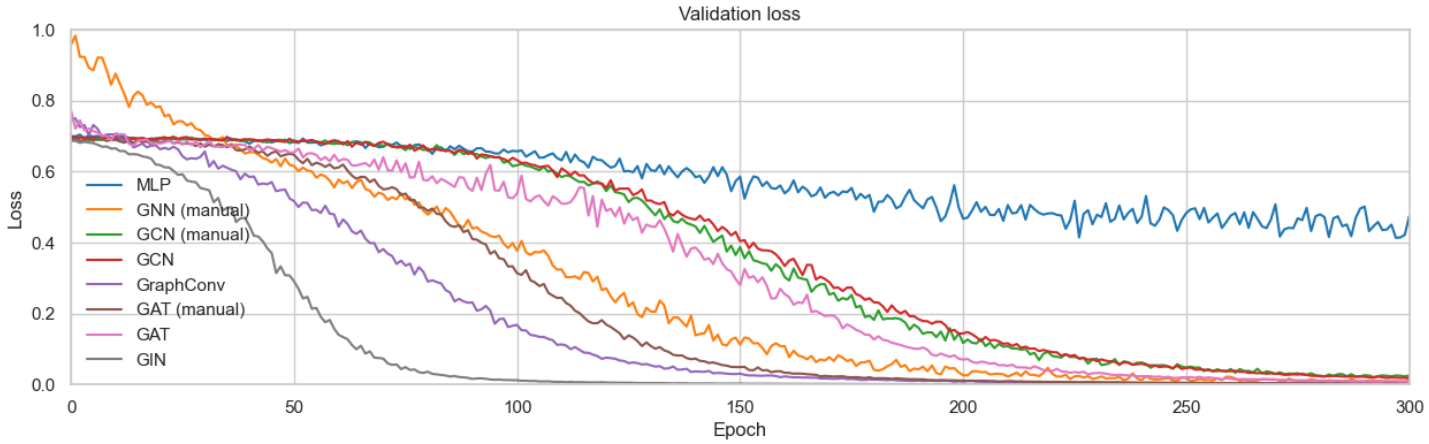


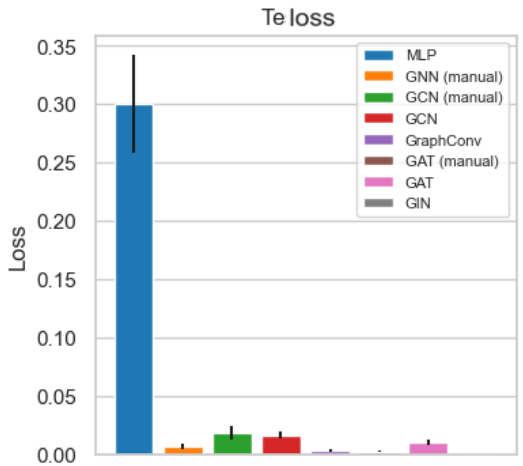


Results (balanced, OHE)

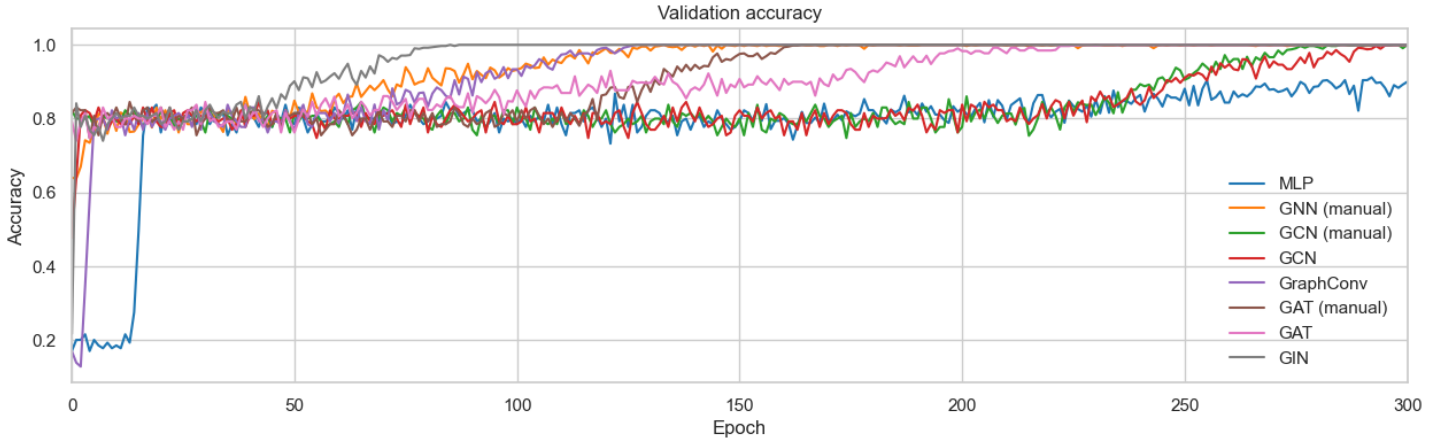


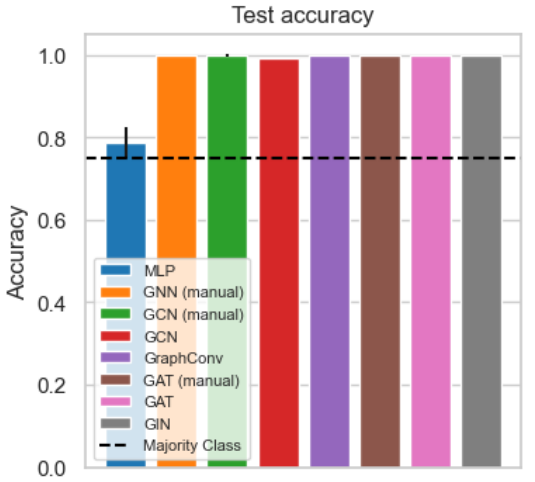


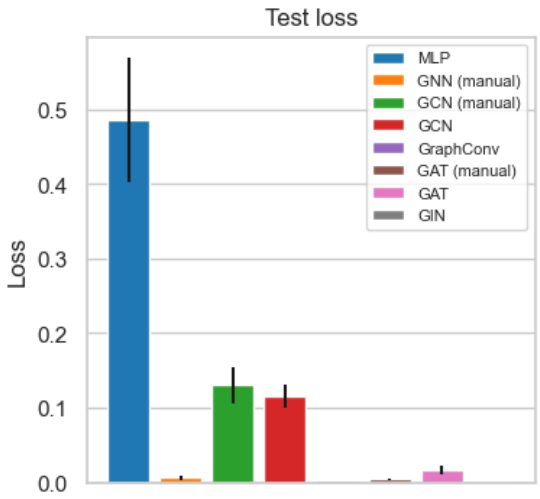




Results (unbalanced, OHE)

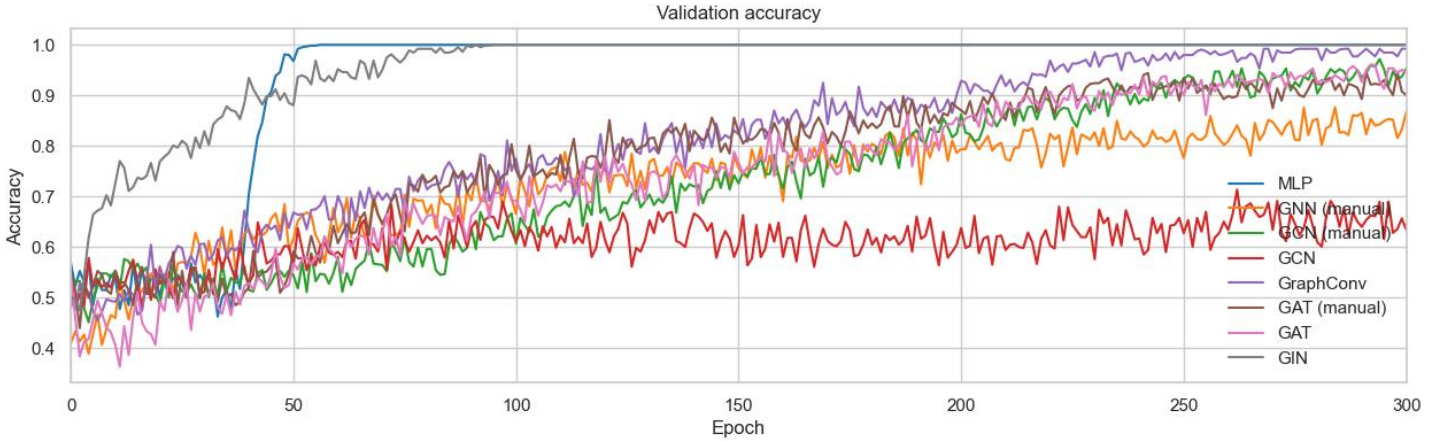


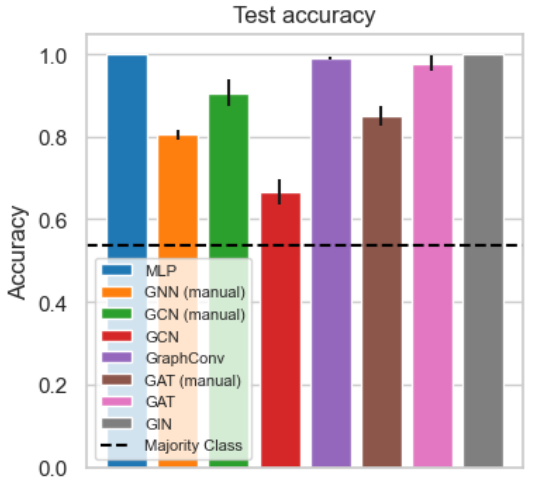


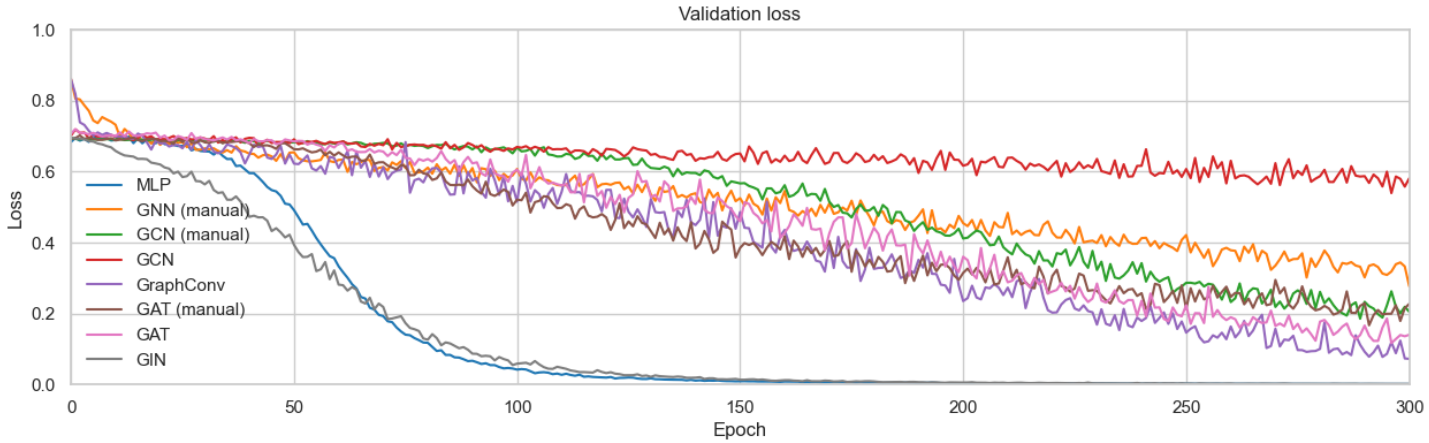


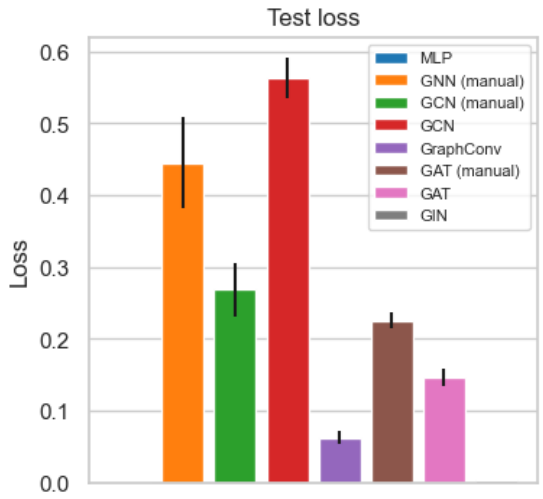
Results (balanced, Node2Vec)

POLITECNICO MILANO 1863



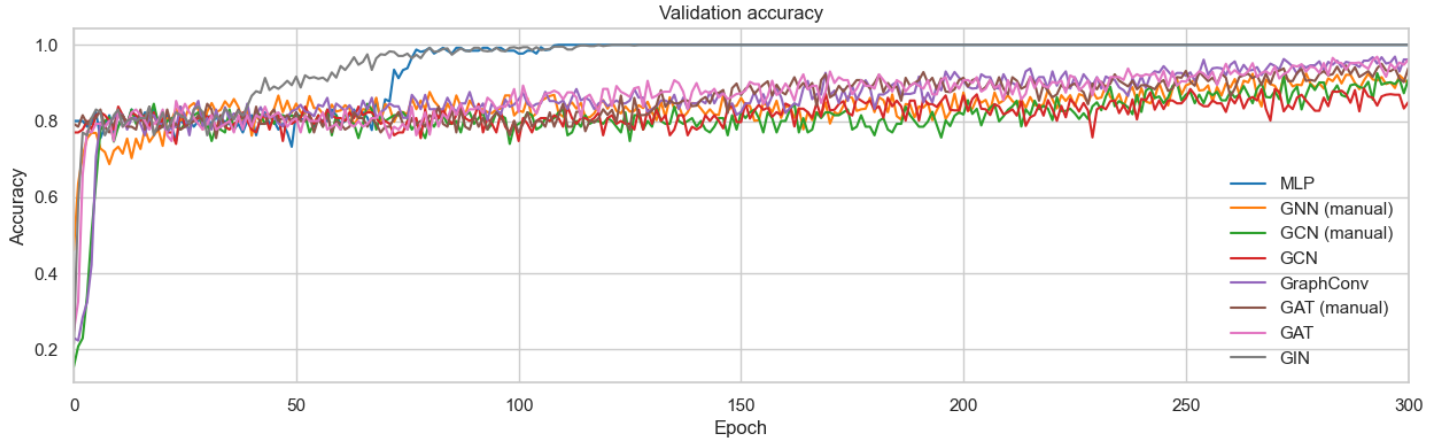


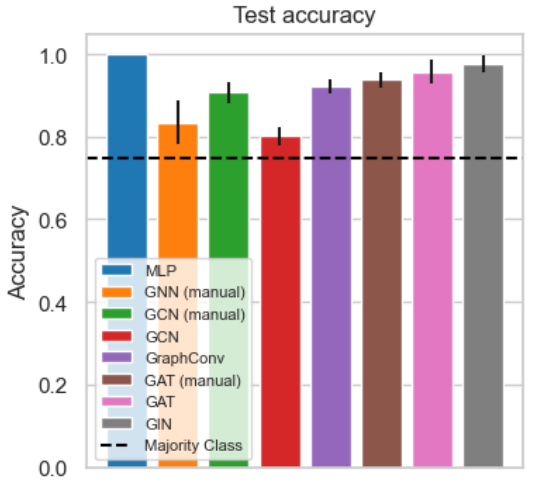


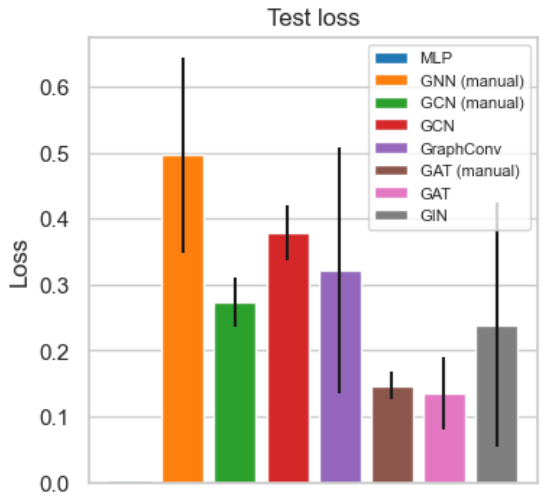


PhD DADS

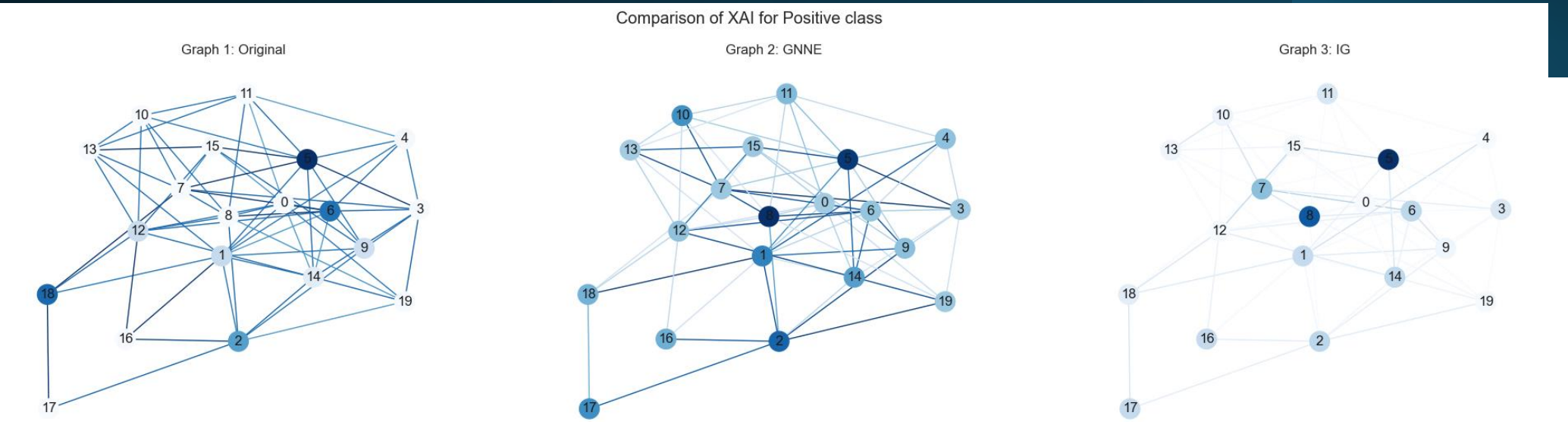
Results (unbalanced, Node2Vec)

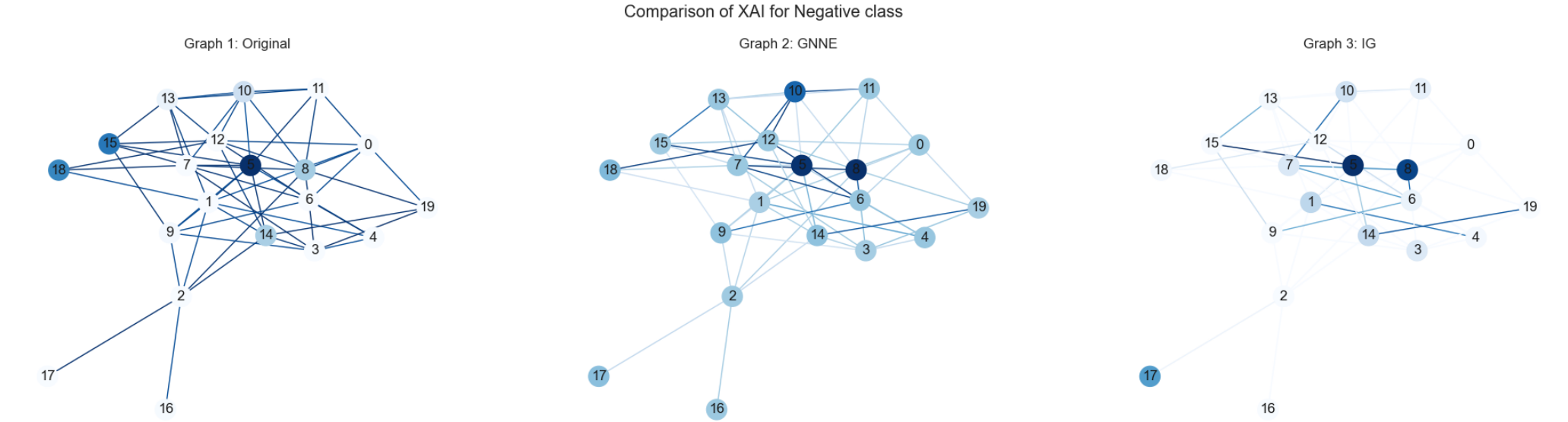






Results (Explainability, GCN)





Both methods do not contradict one another. They highlight the most influential nodes and edges with the largest impact on the prediction.

Results

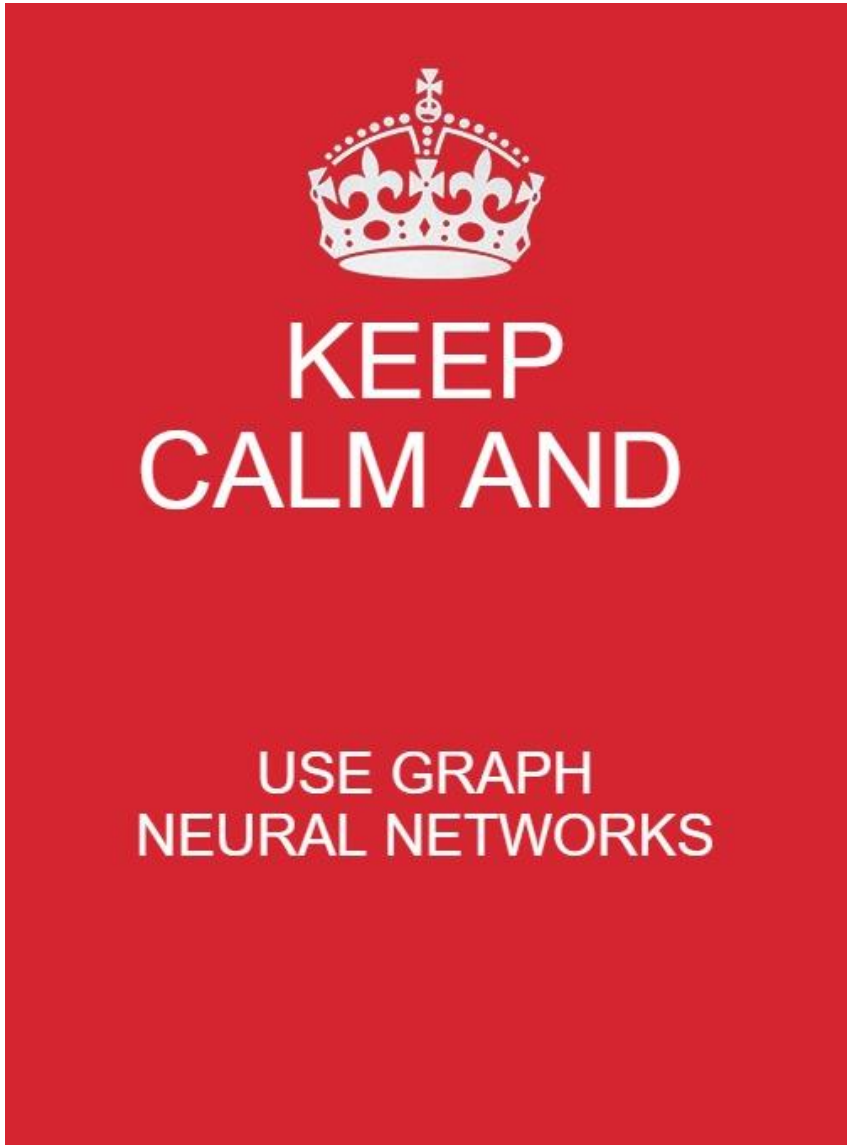
Observations:

- Among all the covered models, GIN manages to consistently classify the graphs, as it generalizes the WL test and hence achieves maximum discriminative power among GNNs (can distinguish graphs which are distinguishable by WL test <- graph isomorphism problem)
- In the case of using the initial features or features from Node2Vec, a sufficient performance is achieved without considering the topology using MLP model (see stackoverflow)
- The addition of simple one-hot-encoding of node indices as features drastically improves the performance for all the graph models
- In most cases the extended models of GNN, in particular, attention-based models, increase the overall accuracy wrt. basic models
- GNNE and IG, mainly, focus on the same nodes/edges for a given graph, but the score values vary

Remarks:

- For the implementation from scratch, the usage of scatter operation substantially increases the time efficiency of the model as compared to the straightforward matrix manipulation.
- Although node embeddings by Node2Vec improve the performance, it would likely require tuning of p and q parameters to achieve the effect of OHE features.
- A discrepancy between the “from scratch” and “off-the-shelf” models is attributed to different initialization of the parameters, randomness in batch sampling, minor differences in implementation and the usage of the edge weights.

Thank you for your attention!



POLITECNICO
MILANO 1863

PhD DADS