Homework 4

Gabriele Matini, 1934803

The objective is to explore the applications, flexibility, generalization, and explainability of Graph Neural Networks (GNNs). The goal is to evaluate how GNNs utilize both the features of individual nodes and the information exchanged with their neighbors through message passing mechanisms.

We will also examine three distinct types of GNNs:

1. Graph Convolutional Network (GCN): it applies convolutions directly on the adjacency matrix of the graph to aggregate and update node features. In particular, the normalized adjacency matrix with added self-loops (so that each node's features is updated also with its own features), is multiplied with the matrix of node's features at layer i and again with the learnable layer i weights matrix, before applying some activation function. Normalization of the adjacency matrix is applied to avoid nodes with many neighbours to have high-magnitude feature updates.

2. Graph Attention Network (GAT): it incorporates an attention mechanism to assign different weights to neighboring nodes, aggregating their information through multiple attention heads. The attention is calculated for each neighbour by multiplying the learnable weight matrix with the feature vector of the node, the neighbour's feature vector, concatenating the two results and then multiplying a learnable vector with the concatenated result and applying an activation function. Finally, we can have multiple versions of the learnable vector and matrix, getting multiple attention heads. The head results can be combined thanks to concatenation in the first layer of our architecture, and averaging in the second layer for dimensionality reduction.

3. GraphSAGE: this network introduces a sampling-based approach, where a fixed number of neighbors are randomly sampled to gather and aggregate information for high scalability. The sampled features are aggregated through some function (mean, sum or max).

It's also important to note that, for any task and any model doing the task, the whole process of training, testing, validating and recording the loss and validation evolution of the model is encapsulated into an "Experiment" class. This class is also helpful to save and load later the trained models.

1 Node Classification Task

The utilized dataset is the Cora dataset, a dataset composed of 20708 nodes, little more than 10000 edges and 1433 features per node. Finally, every node can belong to one of 7 classes. We first work on this dataset in order to remove outliers which constitute noise: this is done by converting the dataset into a networkx representation and then removing all the nodes that are not part of the largest connected component. This reduces the number of nodes only by little more than 200 and helps the GNNs learn actually useful information by removing small remote connected components or isolated nodes.

1.1 The GNNs

The GNNs are all constituted of two convolutional layers different for each type of GNN, so that we have an input layer, a hidden layer, and an output layer for each GNN. A relu is applied at the end of the first layer and a softmax at the end of the second. Finally, a dropout layer helps with regularization.

1.2 Hyperparameters Tuning

A series of hyperparameters are chosen for the fine-tuning phase, including number of heads for GAT and the aggregate function choice for GraphSAGE. We also choose the number of epochs to have an idea of a good number of epochs for training to prevent overfitting. The chosen model is the one that has the highest validation accuracy at the end of the training. We thus define a fine-tuning function that automates this process.

1.3 Training & Test Results

After the fine-tuning phase, we train the models with the suggested hyperparameters and test the results recording metrics such as precision, accuracy, f1-score and recall. From the training history graphs, GraphSAGE looks as if it has a slower or more difficult convergence; however, overall the testing reveals that all models perform somewhat similarly. The graphs of the training history are available in the ”NodeClassification” folder.

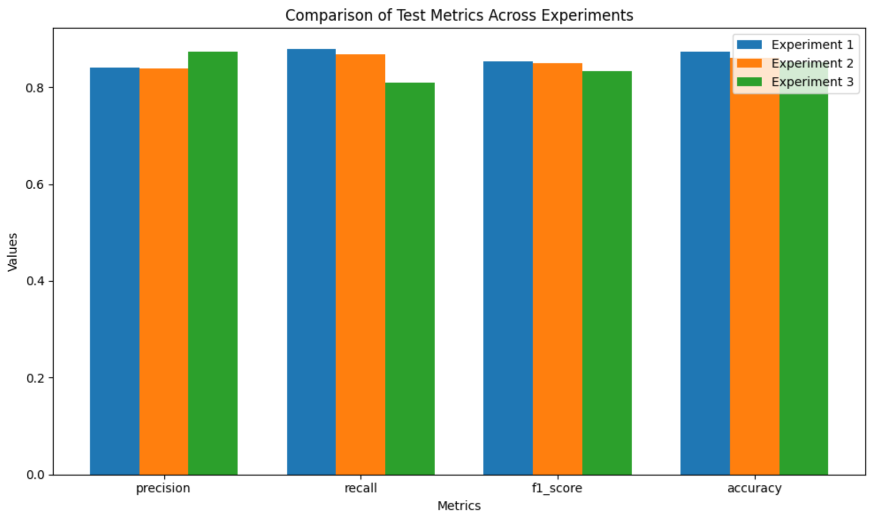
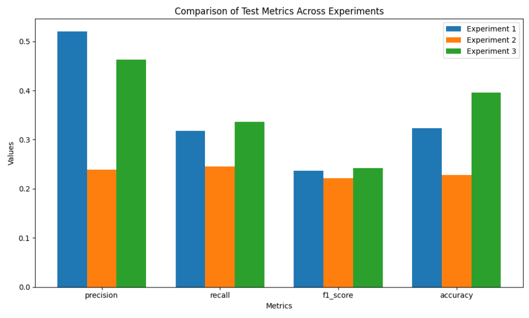


Figure 1: The graph reveals all the metrics for all the models: in blue we have GCN, in orange GAT and in green metrics for GraphSAGE. The models behave similarly: all metrics result to be above 0.8 with some oscillations.

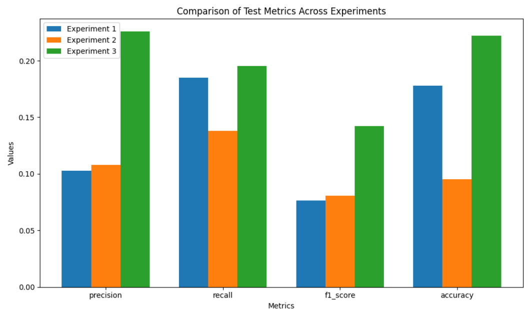
2 Generalization to other datasets

We try now to test the GNNs on two other different datasets: PubMed, which consists of around 20000 nodes, 90000 edges, less features (500) and less predicted classes (3), and CiteSeer which is a dataset that on paper looks much more similar to the Cora dataset for number of nodes, edges, and classes, except for the number of features (almost 4000). The idea is to understand the generalizability of the GNNs with respect to new datasets, so we avoid retraining newly defined GNNs and use the old model with a few tweaks. To begin with, we have the problem that PubMed has less features than our input layer can take, and at the same time CiteSeer has more. Thus, the problem arises of having to change something in order to solve this problem. Secondly, we also have the problem of the fact that the output layer does not match the number of classes anymore, so we'll need to change the output layer to fix this. Three solutions were tried and they all offer some insight into the generalizability to other data sets from the GNNs:

1. Apply zero padding on the PubMed dataset and apply dimensionality reduction on CiteSeer with PCA. This first experiment solved the different input problems and avoids retraining, but the performance significantly degrades.



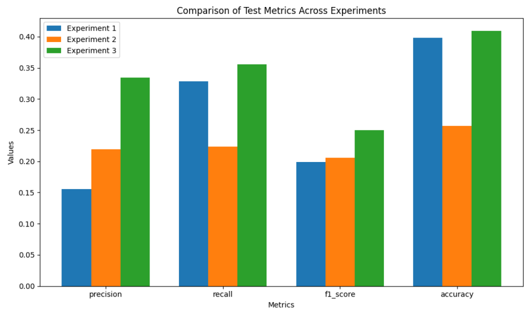
(a)



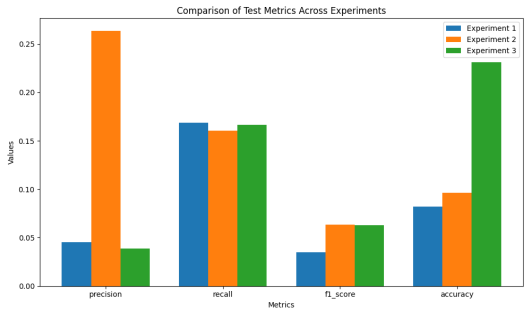
(b)

Figure 2: Performances of the three saved models on PubMed (a) and CiteSeer (b) datasets. Looking at the accuracy, we can see that the models perform just a little better than a random classifier, with the exception of GAT which suffers the most from these changes. In any case performances degraded significantly.

2. We use a dual approach to the first one: if we do not apply padding and dimensionality reduction to the datasets, we apply them to the weight matrix (or matrices in the case of GraphSAGE) of the input layer. PCA was used to reduce the weight matrix. The idea is that PCA can reduce the learned weights but can try still to preserve information learned by the GNN.



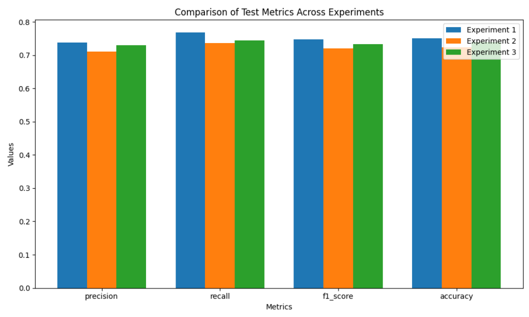
(a)



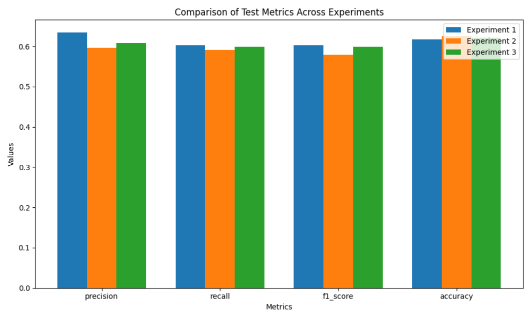
(b)

Figure 3: We get a slight improvement, especially over PubMed. GraphSAGE seems to be fair the best on this particular approach, while again GAT has the poorest performance.

3. The last and most successful approach tries to not change neither the dataset nor the GNNs. Since input and output layer are the problematic parts, we resort to attach a new input layer and a new output layer to our GNN and train only those ones, freezing the inside layers. These layers would in effect encode and decode the input and the output in order to solve the dimensionality problem.



(a)



(b)

Figure 4: Again on figure (a) we can see results on PubMed, while on (b) we see the ones on CiteSeer. Performances are still worse than the ones on the original dataset, but they are now acceptable. GAT is also having the best performances on CiteSeer.

Conclusions Performances differ a lot between PubMed and CiteSeer, although this can be explained by the difference in the number of classes to predict: CiteSeer has 6 classes, while PubMed has 3. We can see how this can affect performances since, a random classifier has a $\frac{1}{6}$ chances of getting the right answer on CiteSeer and a $\frac{1}{3}$ on PubMed. Plotting Cora, CiteSeer and PubMed as graphs, we see a big discrepancy between CiteSeer and Cora: CiteSeer has many singular disconnected nodes, while the largest connected component is quite small in comparison. On the other hand Cora and PubMed have a similar structure, with a singular large connected component at the center, which, especially for the third solution, contributes in getting a decent performance. This discrepancy surely contributes in creating a problem, since the model has a harder time to classify nodes that are not connected to any other node. Couple this with the fact that CiteSeer has a lot more features, and we have elements that can explain why all the GNNs perform worse on CiteSeer. All in all, the models don't seem to generalize well on new datasets, and the only acceptable performances appear in the third solution for PubMed, where we hit around 70% for all metrics.

3 Link Prediction

We define the link prediction task as a binary task predicting whether a certain edge exists or not. For this purpose, we define a dataset by extracting all the positive edges from the cora dataset and generating a number of negative (labeled as nonexisting) equal to the number of edges for the validation and test datasets. We then expand the Experiment class to edge prediction to include the usage of the new dataset pieces, the calculation of the ROC-AUC metric, and a better loss criterion for our purposes, binary cross-entropy. The GCN, GAT and GraphSAGE classes are redefined as GCNEP, GATEP AND GSAGEEP for edge prediction task, and they include the possibilities of doing the dot product between the node embeddings or to concatenate the vectors and pass them through a simple multilayer perceptron. The fine-tuning phase is the same, with the exception that we also have a parameter to use concatenation or dot product. So we can distinguish between two phases: in the first one, the GNNs produce the embeddings, and in the second these embeddings are used for link prediction. The quality of the predictions is thus related to the quality of the produced embeddings.

3.1 Training & Testing

During training we generate randomly the negative edges for the train set (getting a balanced train set), an approach that provides more diversity to the negative sampled edges. The fine tuning also reveals that none of the models uses concatenation, and in fact sometimes concatenation doesn't even work (the model doesn't learn) and other times it doesn't train enough, suggesting that different architectures may have had a better use of concatenation followed by a feed-forward layer. We also plot ROC, AUC and the metrics in the testing phase:

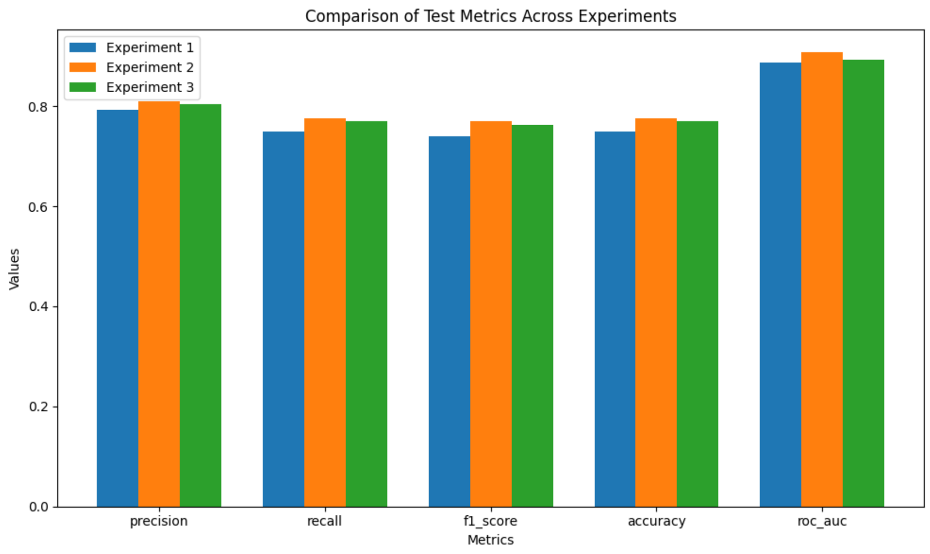


Figure 5: The best metric score is by far the ROC-AUC, which usually surpasses 90%. This indicates that the models are very good at guessing positives.

The confusion matrices suggest that all the models have a hard time distinguishing between true negatives and false negatives (usually 300 negatives over 500 are guessed right), which means that we don't want to employ our GNNs' architectures in a task where having a low rate of false positives is important. Also, all the models seem to score high in AUC, which is encouraging for tasks where it is highly important to find true positives. Social networks might make good use of these models, for example to take a social network's graph structure and recommend possible friends or interactions to other users, to uncover new potential relationships. Recommender systems might take advantage of our models by being able to predict that a user that has bought many of a sequence of similar elements might also want a new element connected to the first ones. Knowledge graphs could also use the GNNs to predict some subclass or, generally speaking, filling in missing links between entities. Even predicting outbreaks of epidemics and their spread might make use of these GNNs, since it is more important to predict all the real spread possibilities, however caution should be ensured with

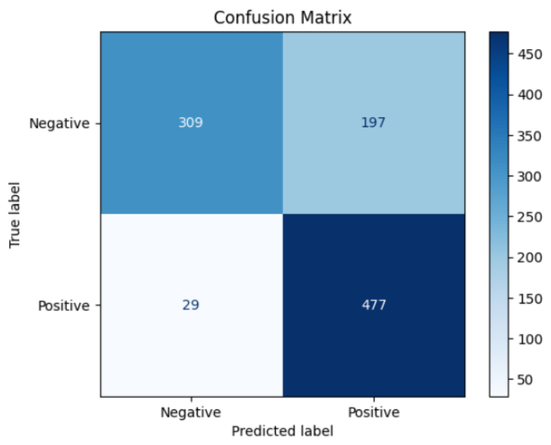


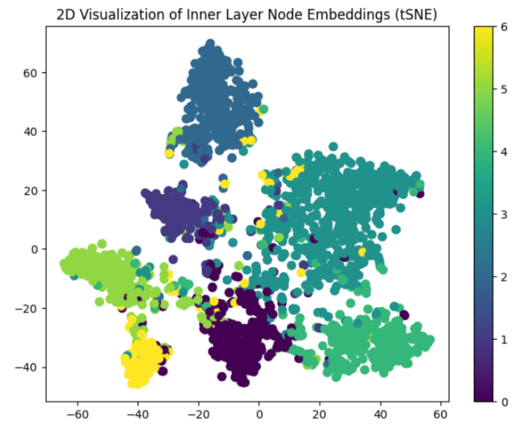
Figure 6: Confusion Matrix: as we can see the model is good at predicting true positives but gets wrong around 40% of the negatives. This pattern arises for all tested GNNs.

respect to costs of shutting down possible wrongly predicted positives. However, many other critical tasks such as fraud detection, criminal enterprise connections prediction, legal evidence analysis or medical diagnoses cannot make use of these architectures, because having a low false positive rate is crucial (we don't want to arrest the wrong person, give a wrong medical treatment, deny a transaction or a loan, and so on...).

4 GNN's Embeddings

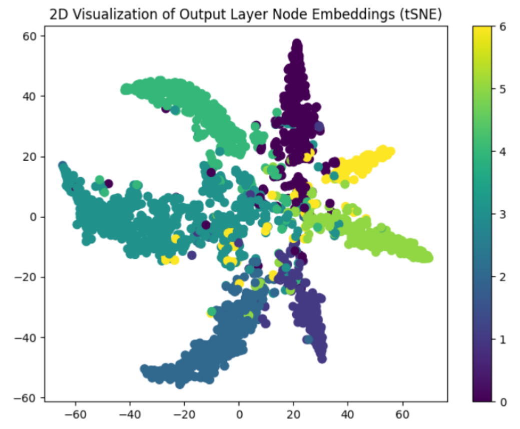
Next, we explore the embeddings generated by the GNNs, to understand how exactly the models cluster these embeddings and how they learn to cluster the nodes. To do this, we wrap the model in a class that calls only the first layer of the GNN: this is how we get the inner layer embedding. We also decided to explore the output layer embeddings to have a sense of the evolution of the embedding's processing overall. We also generate embedding for the link prediction task to see if there are meaningful differences. Both PCA and T-SNE were applied, however T-SNE seems to award the more meaningful plotting results. As we can see, the node classification embeddings seems to

```
Visualize embeddings of GCN
Plotting inner embedding
```



(a) The classification seems to follow a "star" pattern, however some points are still clearly not well clustered.

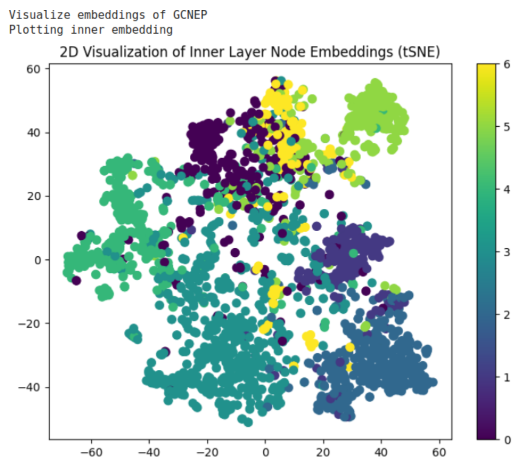
```
Plotting final embedding
```



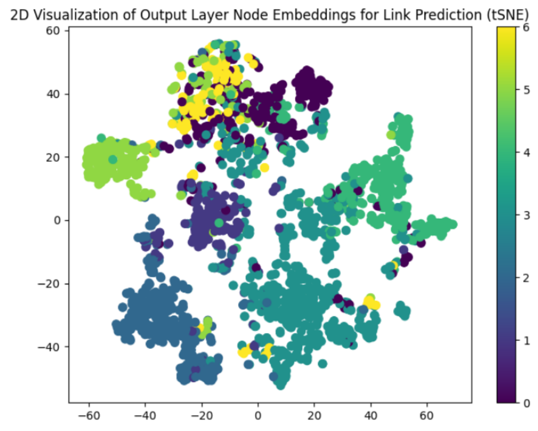
(b) In the final embedding we can see the star pattern emerging more clearly with less misaligned points.

Figure 7: Embeddings of inner layer of GCN and output layer of GCN. Other models' layers are present similar patterns.

follow a star pattern. This is not the case for the link prediction embeddings, although the resulting embeddings seem to still form meaningful clustering patterns:


```
Visualize embeddings of GCNEP
Plotting inner embedding
```

(a)



(b)

Figure 8: Embeddings of inner layer of GCNEP and output layer of GCNEP. Other models' layers are present similar patterns. All the embeddings show a blended, well-structured pattern, likely reflecting the fact that the GNNs can learn the relationships between nodes through message passing.

5 Node2Vec Algorithm

Node2Vec is a machine learning algorithm that generates node embeddings differently than a GNN would, by trying to understand the graph's structure through random walks. The random walks are biased thanks to the choice of two parameters "p" and "q", that regulate how likely the algorithm is to go back to a previously explored node (favouring a BFS search) or to move further away (favouring a DFS search). The random walk is treated as a sequence of nodes to analyze one by one as central nodes: a "window" of nodes in the random walk is used to embed the central node and the embeddings tries to maximize the probability of having as neighbours the nodes in the window given the central node. Our objective is to evaluate Node2Vec on our link prediction and node classification tasks. To do this, we first regenerate link prediction training data, adding negative samples to the training data too. We then define the function to generate the embedding, which does the actual training for Node2Vec, and a fine tuning function that occupies itself with both training and evaluating for the two different tasks. The idea was initially to do two different fine-tunings for the two tasks, but actually fine-tuning the embeddings for the first node classification task gets similar results to produce embeddings that capture in a good manner the structure of the graph, and as such they are ok also for link prediction. In order to have a fair comparison with respect to the GNNs, we replicate as much as possible the same steps that the GNNs do: for link prediction evaluation, after the embeddings are produced we apply dot product to them, and for the node classification we fit a logistic regression algorithm to the embeddings. Finally, we use the same evaluation techniques applied until now, with plotting of the metrics, ROC curve and confusion matrix for link prediction. It's important to note that Node2Vec is very slow to train, and as such there were problems during the fine tuning phase and the only way to solve them was to cut back on the number of hyper-parameters. As such the tried hyper-parameters were p, q, batch size, epochs and dimension of embedding. The fine tuned embeddings are provided in the Node2Vec folder.

5.1 Node2Vec testing

Even if the training is slower, the results are encouraging: node2vec scores a bit worse with respect to the GNNs on node classification and way better on link prediction, achieving even 1 for recall for some embeddings. It is expected that the GNNs score better on node classification, since their message passing and feature centric approach would allow to better understand the class of a node, while Node2Vec needs to rely on the graph's structure, which is what it learns. This, however, gives Node2Vec an edge on link prediction: by pointing out the fact that Node2Vec is an algorithm specifically made to understand a graph's structure, and as such it will score better on tasks that require a ML model to understand if the graph should have a certain link or not, we can see how Node2Vec would score better on a task that aligns itself naturally with the algorithm learning process. However, just like the GNN, Node2Vec too has difficulty with precision: the confusion matrix reveals that still around 200 negative edges over 500 are misclassified. This could at this point reveal a difficulty for any algorithm in assessing that an edge should not exist: in fact the graph is the largest connected component of cora, and as such it might be very connected, and so even Node2Vec will find it difficult to assess that an edge should not be present.

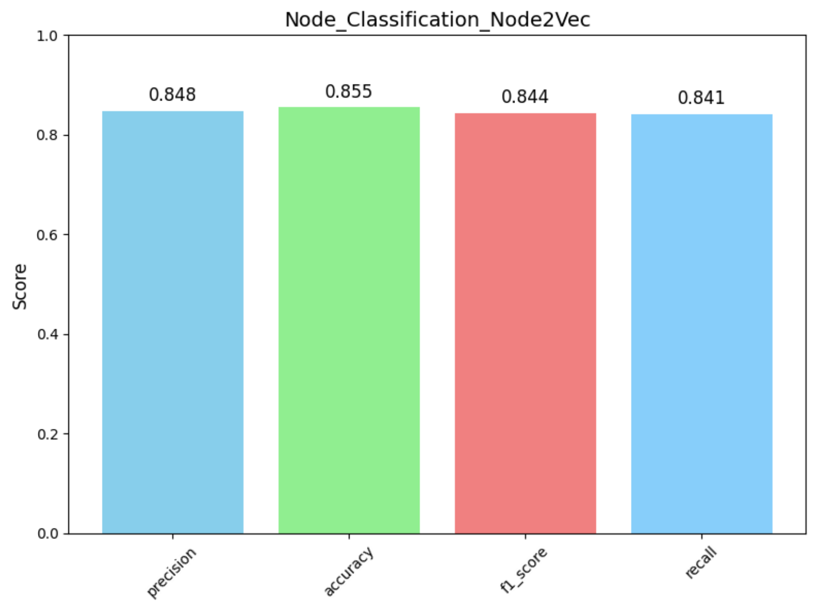


Figure 9: Node classification metrics for Node2Vec embeddings.

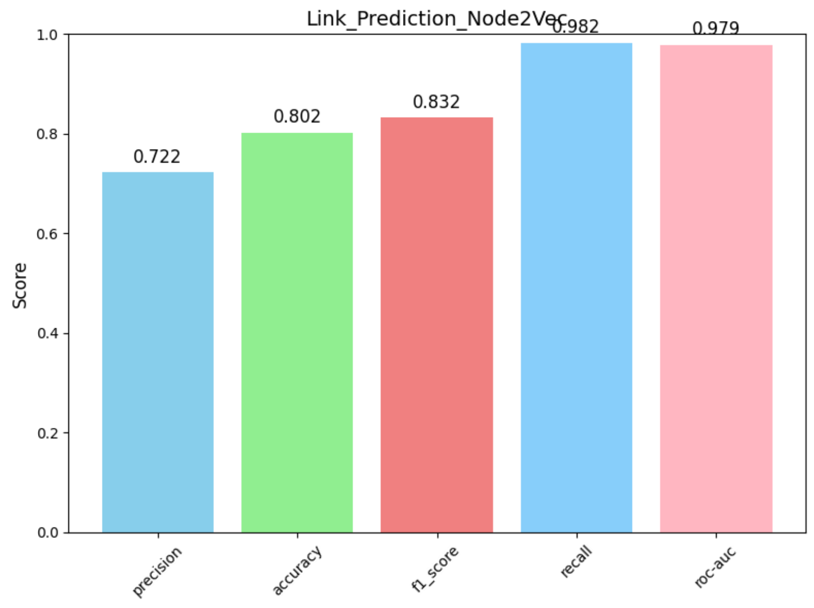


Figure 10: Link prediction metrics for Node2Vec. In the files there is also the link prediction done with the embeddings of node classification, which achieve 1 in recall. However, results are not much different.

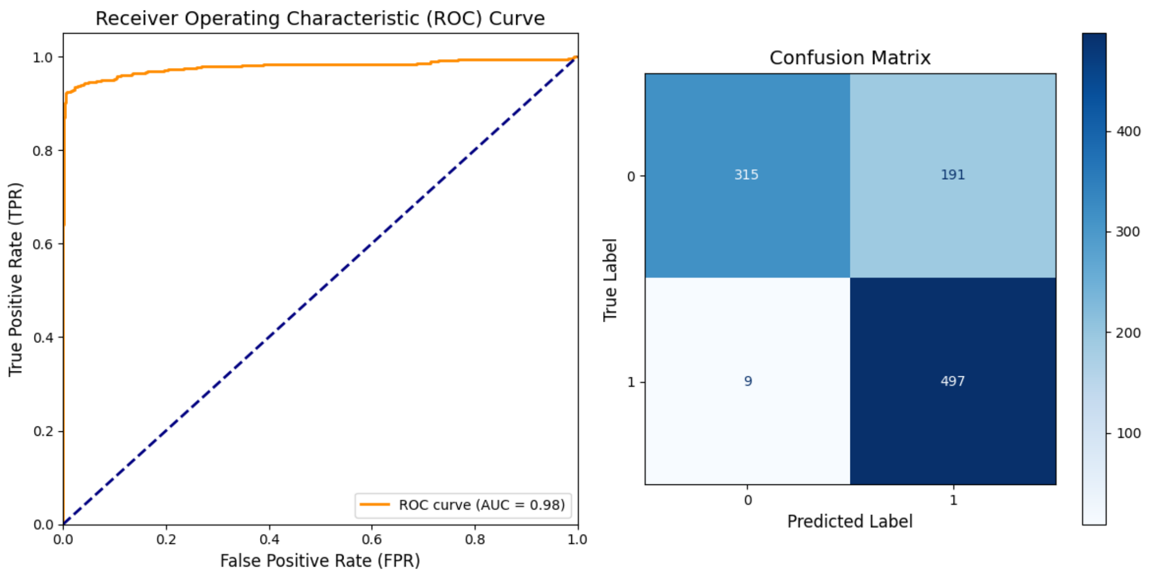


Figure 11: Again, we can still see that even Node2Vec has some degree of difficulty with false positives.

5.2 The embeddings

Node2Vec embeddings have a very different form from the GNN ones: now the points seem to be clustered around bigger and smaller pockets, with the smaller pockets and the majority of points of one cluster being clustered around the biggest pocket:

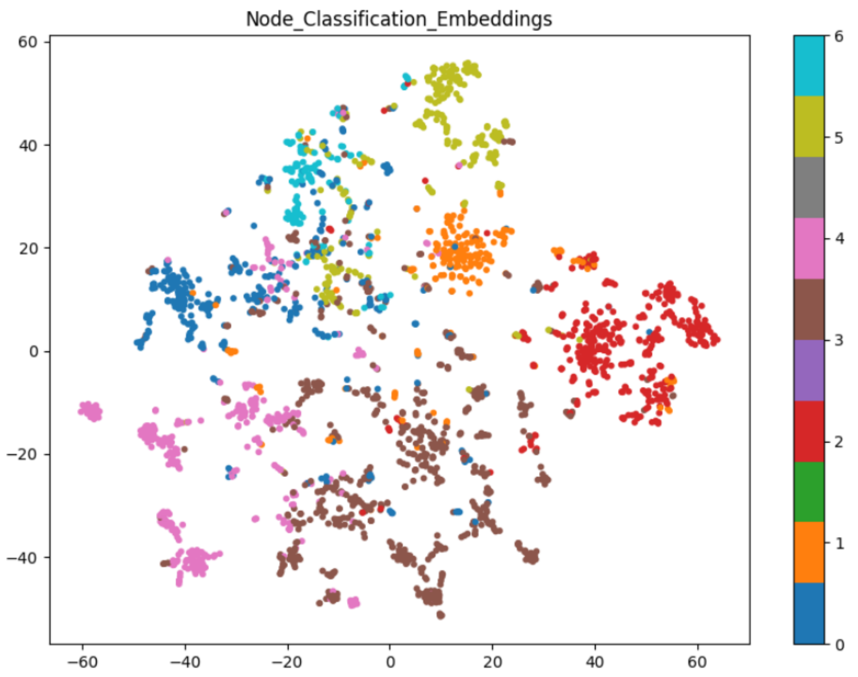


Figure 12: Embeddings of Node2Vec. We can see a pocket pattern in the embeddings. Each node appears to be part of a neighbourhood of equally classified nodes, and the equally classified neighbourhoods appear to be clumped up together. This can reflect a separation between pockets of similar nodes or it could be the result of the choices of p and q. We can also see that at the ”borders” between one class and another we have some nodes of both classes, indicating perhaps some ”bridge” between these pockets. There is less blending compared to the GNNs' embeddings, showing that Node2Vec learns less higher-order information, and more about the raw graph structure, which aligns better with tasks like link prediction. The embeddings produced by the fine tuning for node classification and the one for link prediction are available in the node2vec folder.

6 Explainability

The last section regards explainability of the GNNs' decisions with respect to node classification: we want to understand what aspects of the graph contributed to some node's classification. To do this we use the GNNExplainer class. GNNExplainer works by learning a mask over the graph to both features and edges to understand which are the most impactful on the model's prediction. It starts with a random edge/features mask and performs gradient descent to learn the best mask that approximates what the model was focusing on when it made the prediction. To use the class correctly we need to adjust the forward method of our models, so we wrap them in a wrapper class that has the expected forward method and calls the underlying old model class. We then use a function that, for each node in a random list of nodes we will scrutinize, visualizes the most influential edges of the learned mask.

6.1 Analyzing specific nodes' predicitions

Following there are some specific nodes commented upon. It's important to note that these nodes have been explained for every architecture type, although not all the screenshots of all the nodes will be reported here, especially if the explanations are similar from GCN to GAT, to GraphSAGE. However, all the versions are reported in the "Explainability" folder. It's important to note that predictions between models change slightly: this is explained by the different ways in which GNNs get the information: GCN considers neighbours and features uniformly, GAT uses attention mechanisms to focus only on specific connections and features deeemed important, while GraphSAGE results may be altered by the random sampling of the neighbours.

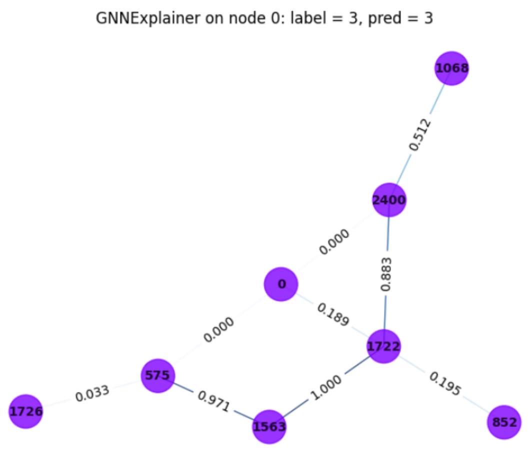
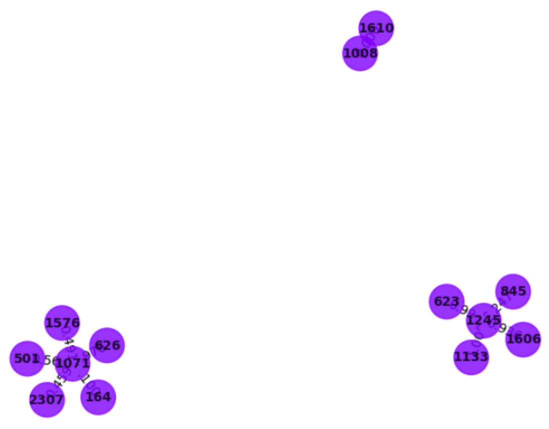


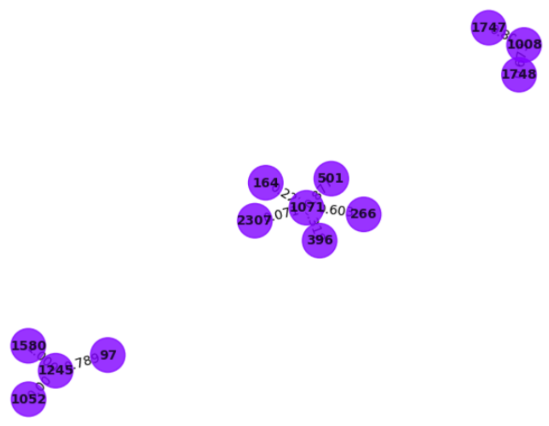
Figure 13: Explanation for node 0 of GCN: the node seems to be immersed into a neighbourhood of same label nodes, fact confirmed also by the fact that the other models take the same (or almost the same) neighbourhood to make the prediction. The edges' weights vary between the different models' though, which indicates either a shift in the gradient descent of GNNExplainer or the models choosing to focus on different features. The fact that same neighbourhoods are present among different models might suggest that homophily (tendency of same nodes to be together) played a role in node 0's classification.

GNNExplainer on node 681: label = 2, pred = 2



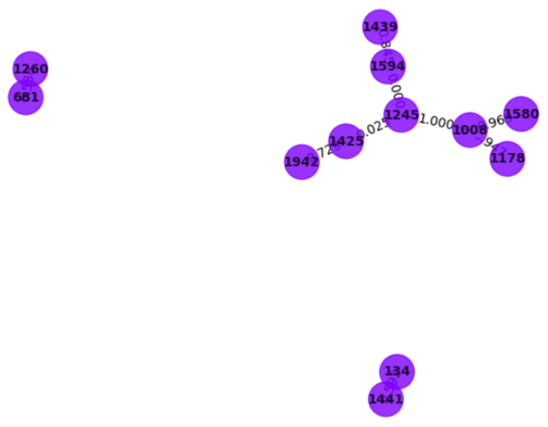
(a) GCN subgraph of node 681

GNNExplainer on node 681: label = 2, pred = 2



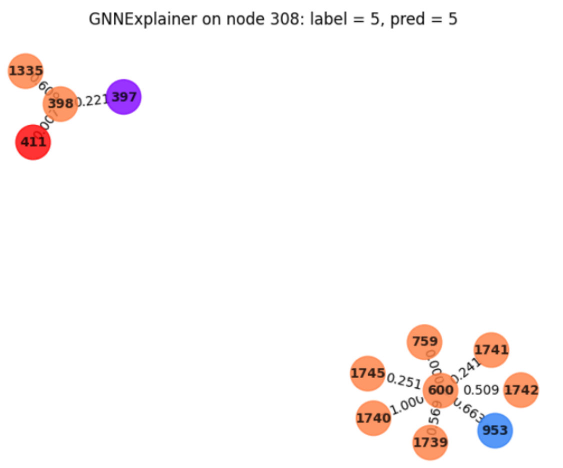
(b) GAT subgraph of node 681

GNNExplainer on node 681: label = 2, pred = 2

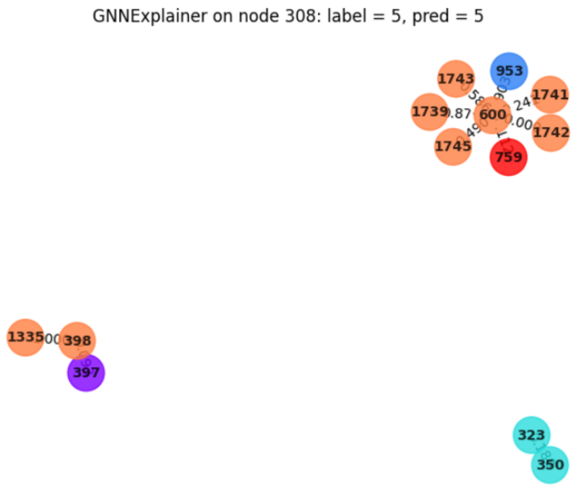


(c) GraphSAGE subgraph of node 681

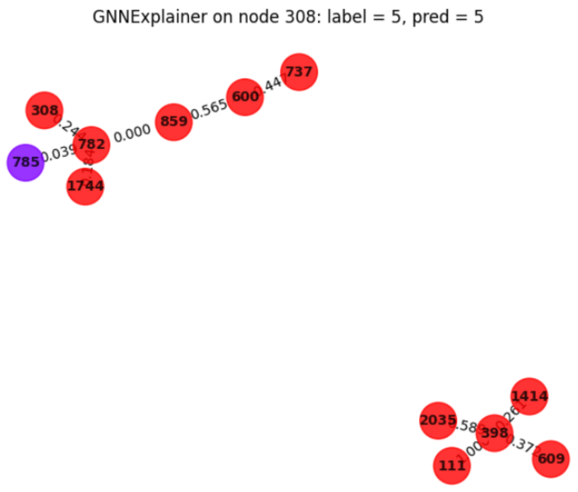
Figure 14: We analyze node 681 to show a case in which the subgraph structure between models varies a bit. We can still see some resemblance between the subgraphs, like the fact that node 1245 seems to be an important hub for 681's classification, but overall nodes and links change.



(a) GCN subgraph of node 308.



(b) GAT subgraph of node 308

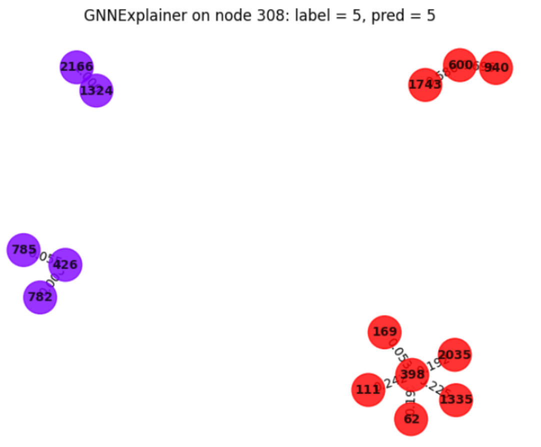


(c) GraphSAGE subgraph of node 308

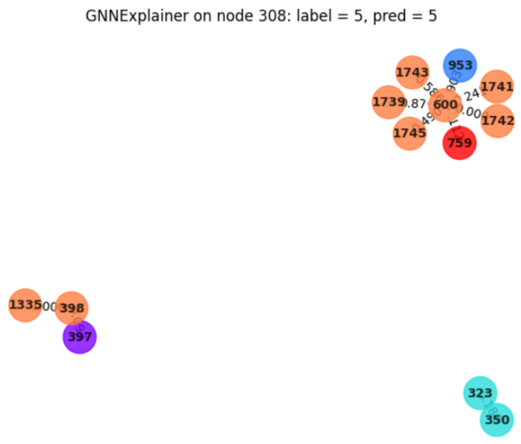
Figure 15: This time we can see how a diverse neighbourhood contributes to a node's classification. This time node 308 itself is not present, which only suggests that none of its immediate connections are important for the predictions. However the mask is still applied on its neighbourhood, revealing that node 308 is probably a border node between different clusters of nodes. It's interesting to see that the models predict the other nodes of being of different classes from each other: GAT and GCN predict node 759 as belonging to different classes, while GraphSAGE is looking at only two different types of nodes being present, maybe due to GraphSAGE's sampling of the subgraph that introduced less diversity.

6.2 Usage of AttentionExplainer

Other than GNNExplainer, there was the possibility of trying an attention based algorithm to explain GAT. What AttentionExplainer does, is use the attention weights produced by GAT in order to weight the edges of the final explanation. The results are similar, but all in all not much more informative than the normal GNNExplainer. Still the AttentionExplainer is tailored for attention based GNNs, and as such might offer a more precise explanation.



(a) GAT subgraph of node 308 using AttentionExplainer.

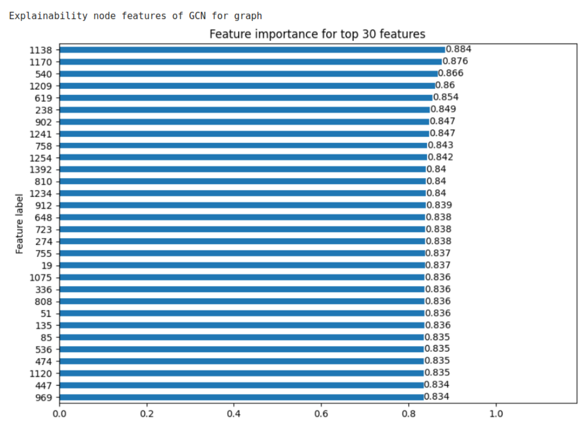


(b) GAT subgraph of node 308

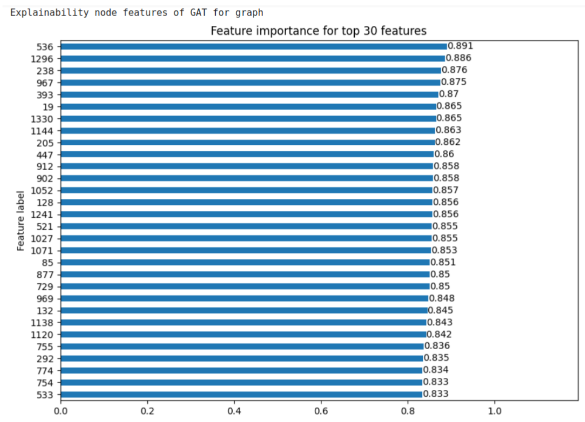
Figure 16: Comparison between prediction made for node 308 using AttentionExplainer and GNNExplainer. Nodes 600 and 398 still seem to be in either case important to the final prediction, however, the choice of the overall neighbourhood seems to have shifted somewhat, indicating that the attention weights were in a slightly different place than what the GNNExplainer guessed.

6.3 Feature importance explainability

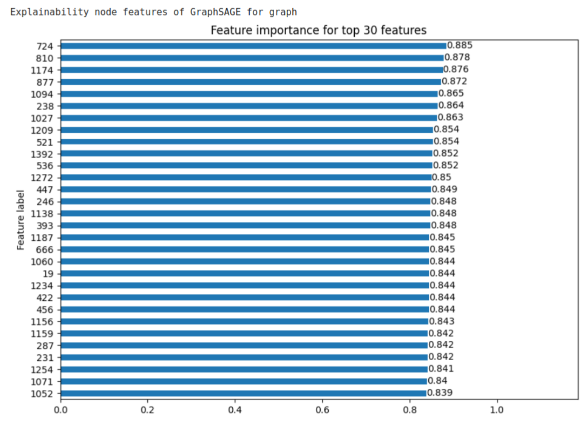
Now, a GNN does not just get information through message passing, but it also considers the node's features in order to predict its label. Thus, the other side of the model's decision to explore is the weight of its features. GNNExplainer has this built in, since it can plot and visualize the most important features for the entire graph and the most important features of a node that defined its predicted label. To get the comparison of most important features over all nodes, we set the attribute of node mask type to ”common attributes”, which means the explainer focuses on global patterns of feature importance rather than node-specific variations. By doing so, we gain insights into the features that most influence the GNN's predictions at a global level. We use then instead the ”attribute” mask to get most influential features for a specific node.



(a) GCN Global Feature Importance

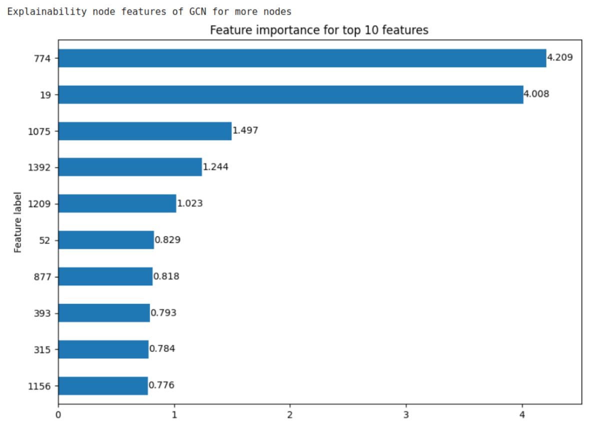


(b) GAT Global Feature Importance

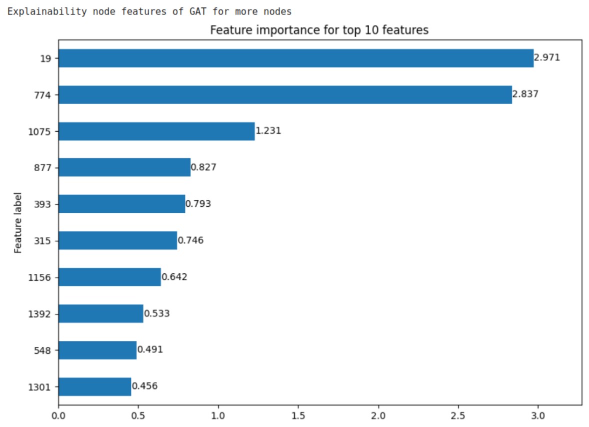


(c) GraphSAGE Global Feature Importance

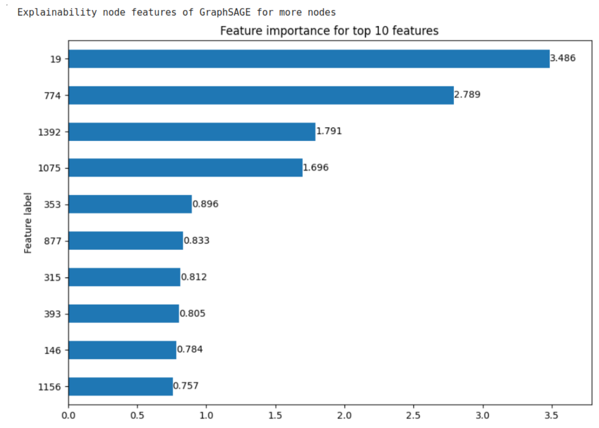
Figure 17: Comparison of top 30 most important features over all nodes of GCN, GAT and GraphSAGE. As we can see, some features like feature 19 are present in the top 30 of all models, suggesting an inherent information value in the feature itself, while many other are present in at least 2 out of 3 models as top 30 features. We can thus find a degree of consistency across different models, which is useful to understand which features are most important to make a node's prediction, regardless of the model (model-agnostic features).



(a) GCN Feature Importance node 0



(b) GAT Feature Importance node 0



(c) GraphSAGE Feature Importance node 0

Figure 18: Comparison of model's feature importance scores for node 0's prediction. Note how we get raw scores instead of the (likely) normalized ones for the global feature importance measurement. We can observe two important facts: first, the models seem to agree on a lot of the most important features, indicating some degree of consistency for what is important in predicting node 0's label. Secondly, we find again feature 19 in all the models, further reinforcing the idea that feature 19 is important for all nodes' predictions. Furthermore, highly scoring features like feature 774 and feature 1075 are present in the top 30 global features of at least one model. Thus, we can find overall a good consistency.