INDIANA UNIVERSITY
BLOOMINGTON

SP23-BL-ENGR-E536

Project Final Report

ENGR-E-536 High Performance Graph Analytics

Prof. Ariful Azad

STUDY THE IMPACT OF GRAPH SPARSIFICATION ON THE PERFORMANCE OF GNNS

Submitted by:
Rahul Saini,
Raghav Chegu Shyam Kumar,
Tarun Krishna Edpuganti,
Deepal Gunda

ABSTRACT

Graph Sparsification refers to the process of approximating an arbitrary graph to a smaller subgraph by removing edges but maintaining the characteristics of the original graph. We achieved significant results in terms of the reduction of computation time while achieving a very similar node classification accuracy of the original graph. A single heuristic is proposed based on Link prediction and contrasted with random edge sampling technique.

- Talk about the problem with existing sparsification techniques
- Propose our solution as an approach with its pros
- Write about GNN's comparision in terms of accuracy
- Talk about the sparsification factors effect
- Talk about the hyper parameters chosen and their effect

INTRODUCTION

Sparsification aims to depict a dense graph by using a sparse graph, while retaining the essential structural features to a reasonable extent. This is feasible for a range of structural properties.

Graph Neural Networks (GNNs) (Kipf & Welling, 2017; Hamilton et al., 2017) have been widely used by the researchers across the graph community, which justifies our decision to choose them for testing the impact of sparsification. We propose a novel sparsification technique based on Link prediction. Specifically, we test the impact of sparsification on the performance of GNN's. GNN's used include GCN, GraphSage and GAT. As a metric to test the performance of each of the GNN's, node classification accuracy and computation time had been chosen. Firstly, we introduce a random edge sampling heuristic used commonly for sparsification and then move on to propose a technique based on Link prediction. We also answer questions such as "Does link prediction accuracy have a significant say in the description of the graph characteristics" based on the results.

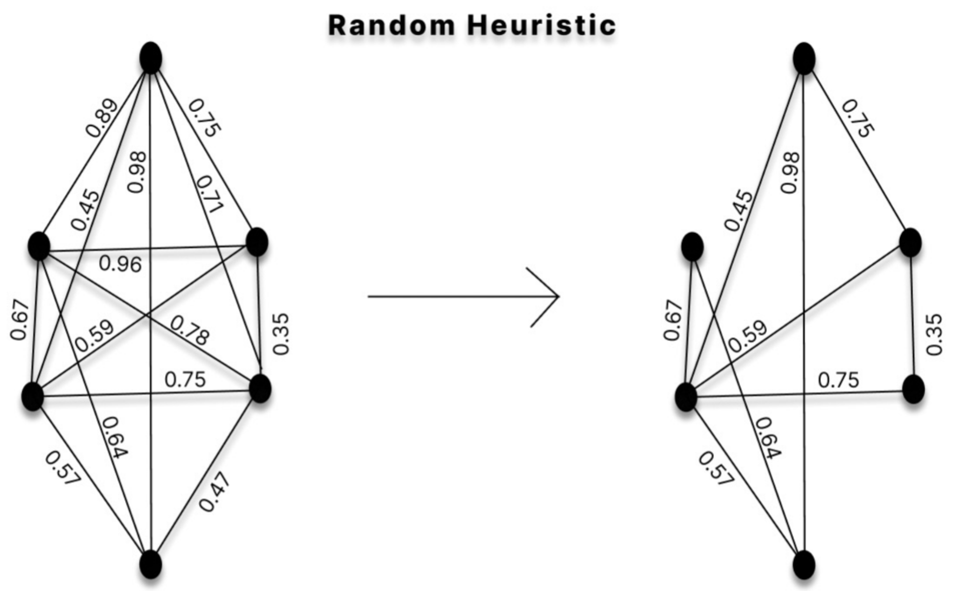
METHOD

Our model comprises three distinct phases, each with a specific objective.

In the first phase, we leverage a Graph Neural Network (GNN) model to generate node embeddings for the input graph. The GNN model takes a graph as input and outputs a node embedding matrix of size (n, ed), where n is the number of nodes in the graph, and ed is the desired embedding dimension. We freeze the weight matrix of the last layer of the GNN model to obtain the node embeddings.

In the second phase, we sparsify the graph using two different approaches: random and heuristic. In the random approach, edges are randomly deleted from the graph. In the heuristic approach, we use a link prediction function that utilizes the embedding matrix generated in phase one to predict the probability of each edge. The link prediction function uses a Breadth-First Search (BFS) approach to ensure that the probability of an edge depends on its neighboring edges. By applying either of these two sparsification methods, we aim to reduce the computational complexity of the model without sacrificing its accuracy.

In the third and final phase, we evaluate the performance of a Random Forest classifier trained on the sparsified and unsparsified graphs. By comparing the node prediction accuracy of the classifier between the two graphs, we can determine if either sparsification method has resulted in a reduction in performance. Overall, our model allows us to efficiently process large-scale graphs while maintaining high levels of accuracy and robustness.



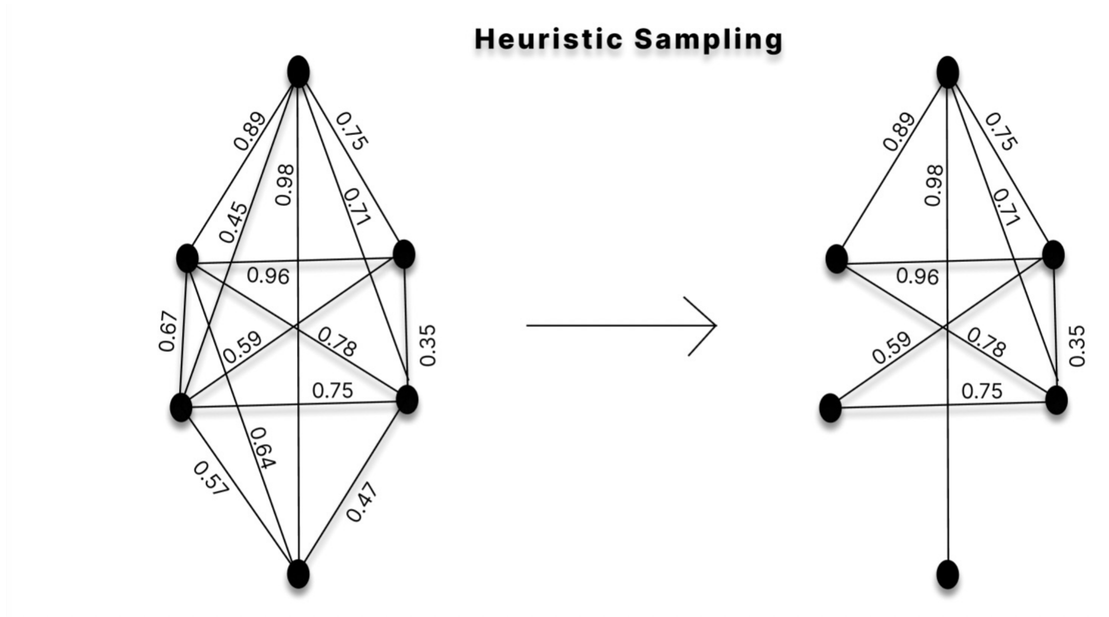
Pseudocode:

Input: num_edges, num_to_delete
Output: edges_to_delete

```
1. Calculate the probability of selecting each edge
  prob = [1/num_edges] * num_edges
```

```
2. Randomly select edges to delete
  edges_to_delete = np.random.choice(range(num_edges), size=num_to_delete,
replace=False, p=prob)

3. Return the selected edges as output
  return edges_to_delete
```



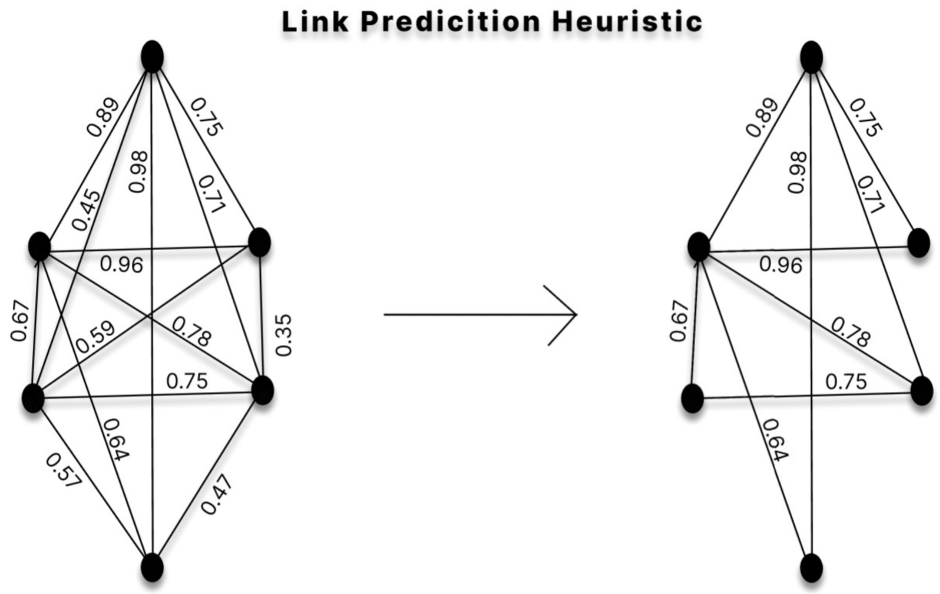
Pseudocode:

Input: num_edges, num_to_delete, prob
Output: edges_to_delete

```
1. Create a list of indices for all the edges in the graph
  edge_indices = range(num_edges)

2. Randomly select edges to delete from the list of indices
  edges_to_delete = np.random.choice(edge_indices, size=num_to_delete, replace=False,
p=prob)

3. Return the selected edges as output
  return edges_to_delete
```



<u>**Pseudocode:**</u>

```
1. Convert the node embeddings to a numpy array and store it in `X`.
2. Create two empty lists `Xd` and `Yd` to store the input data and output labels for link prediction.
3. Iterate over all nodes in the graph:
    a. Get the list of neighbors for the current node `u`.
    b. For each neighbor `n` of `u`, compute the difference between the embeddings of `u` and `n` and append it to `Xd`.
    c. Append a `1` to `Yd` to indicate that an edge exists between `u` and `n`.
    d. Generate negative samples by randomly selecting non-neighbors of `u` using a BFS approach.
    e. For each non-neighbor `nn` selected, compute the difference between the embeddings of `  u` and `nn` and append it to `Xd`.
    f. Append a `0` to `Yd` to indicate that no edge exists between `u` and `nn`.
4. Shuffle the input data and output labels using a random permutation of indices.
5. Split the shuffled data into training and testing sets for different fractions of training data.
6. Train a logistic regression model on the training data and evaluate its accuracy on the testing data.
7. Print the accuracy for different fractions of training data.
```

Note: The BFS approach is used to select non-neighbors of a node to ensure that the probability of an edge depends on its neighboring edges. This means that the negative samples generated using BFS are more representative of the true absence of edges in the graph.

EXPERIMENTAL SETUP

- Using Graph Convolutional networks for the Cora dataset, we freeze the last layer and take those embeddings. Use those embeddings for link prediction using different algorithms.
- From above snippets of code provided, for the Cora dataset, we utilize 3 different GNN models (GCN, GraphSage and GAT). Taking 'GCN' as an example for explanation, 'GCN' is used and the hyperparameters chosen for 'GCN' are learning rate is 0.01 and 2 layers in the neural network. Freeze the last layer and get the node embedding from GCN.
- For the above GCN the accuracy is 0.7440. After freezing the last layer and obtaining the result of node embeddings of the final layer, pass this output as input to the link prediction function which uses 'Logistic Regression'. Using 80% of the data to train and 20% to test, check the accuracy of the link prediction. The accuracy of link prediction using Logistic regression is 0.5167. One reason that we could attribute for such poor accuracy could be attributed to non-linear relationships between input features and the output target variable. As the accuracy for Link prediction using Logistic regression was very low, we employed a different model to see if the accuracy increases for the same node embeddings that we were getting from GCN final layer with no change in the parameters.
- This time around the model used was SVM model. While link prediction we have used SVM for the predictions where we split our data in 80:20 ratio for train and test. From the above snippet we can see that the accuracy while using SVM in making link predictions has increased to 0.8309. One of the primary reasons why SVM has such high accuracy when compared to Logistic Regression is that it handles nonlinear relationships better.
- After using SVM for Link prediction to check if the accuracy can be increased even further, we have used Random Forest method in link prediction. For the link prediction function the input remains the same. Using Random Forest in link prediction, there were hyperparameters like number of estimators = 100, the training and testing split remains the same. As we can see, the accuracy for Random Forest is 1. Reasons for high accuracy could be that it is capable of capturing relationships that are not linear, handling attribute connections handling data that is unbalanced, using ensemble learning to increase precision, and being resistant to data that is noisy.
- In order to make sure that the model is not overfitting, we added a regularizer to the Random Forest model to see if there is change in accuracy. We see that the accuracy dropped to 0.7694. Regularization helped prevent overfitting but to prevent underfitting and loss of accuracy, it necessitates precise hyperparameter tuning and ought to be used sparingly. From above we understood that by employing regularization there is a drop in accuracy which is below the accuracy of SVM.

- So, now we tried to look at the best possible number for understanding what the count for n-estimators should be. After trying random forests with different estimators, we can see the above accuracies. This is when we decided that Random Forest would be the best method that could be implemented for Link predictions because it provides us with the maximum accuracy.
- Parameters for GCN across is constant where we self.conv1 = GCNConv(dataset.num_node_features, 16), self.conv2 = GCNConv(16, dataset.num_classes) and learning rate is 0.01. While adding regularization we considered minimum of 5 samples. For random forest we have n_estimators set to be 100.

Models used in Link Prediction	Accuracies
Logistic Regression	0.516
SVM	0.8309
Random Forest	1.0
Random Forest with regularization	0.7694

Software Versions: All the code has been written in python using libraries such as numpy, pandas, NetworkX, scikit-learn, Pytorch.

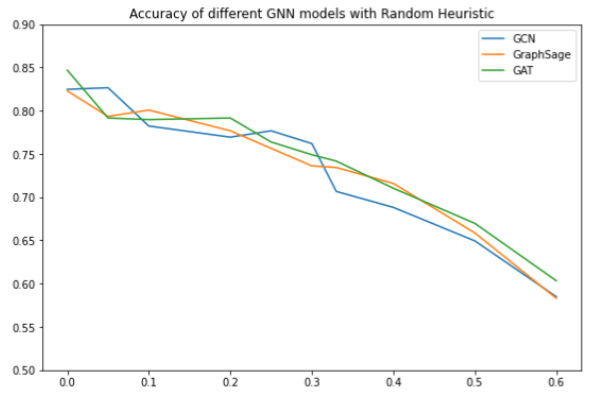
Hardware Platforms: We have used our local systems to run the jupyter notebooks.

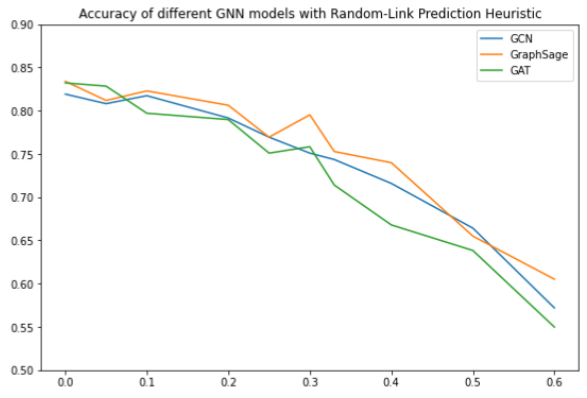
Dataset: The cora dataset from torch_geometric.datasets was chosen to conduct all the experiments during our study. This is a full citation network dataset in (Bojchevski, A et al 2017). Nodes represent documents and edges represent citation links.

Dataset link: Cora full

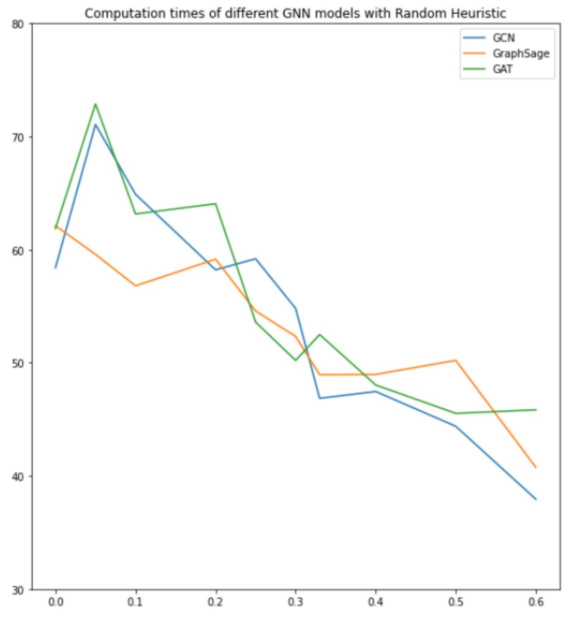
RESULTS

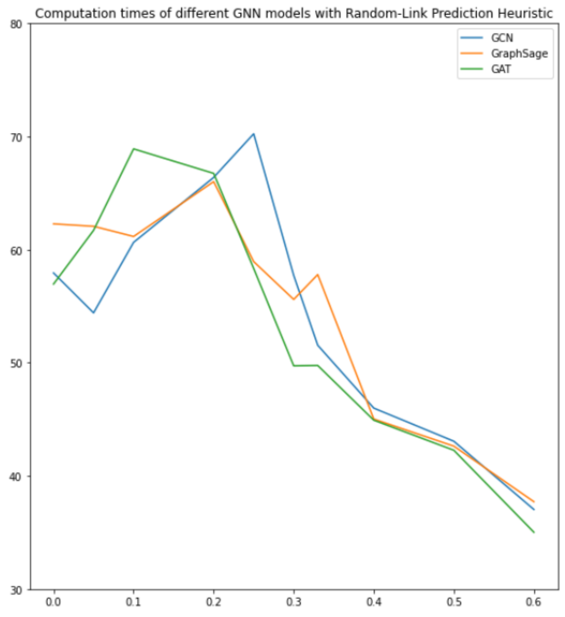
Utilizing the cora dataset and running multiple configurations for the same, we achieved some good results in both the heuristics across the different GNN models (GCN, GraphSage and GAT). The below graph showcases the Sparsification % vs Accuracy for both the heuristics -





It can be seen that across both the heuristics that there isn't a significant difference in the accuracies but there is certain variations with respect to the different GNN models.



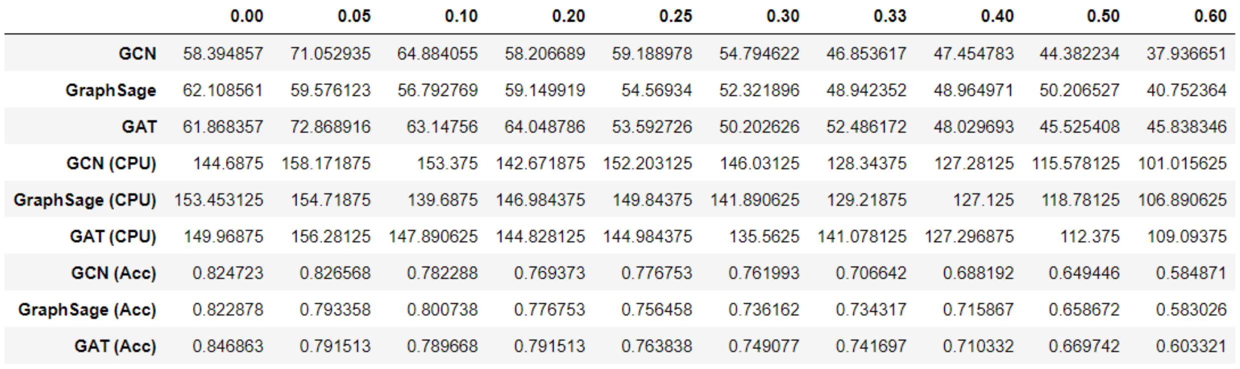


It can be observed that there is a significant decrease in computation times in both the heuristics with increase in sparsification. This is expected but it can also be seen that there is an initial rise in computation time and then a decrease. The behavior could be attributed based on the system computational capacity and could vary between systems.

Random Heuristic (Random deletion of edges) -

- All the GNN models have better accuracy than the others at different sparsification levels with GraphSage and GAT doing slightly better and more consistent.
- There is an initial rise in time which is unexpected but could be attributed to other factors (system computation).

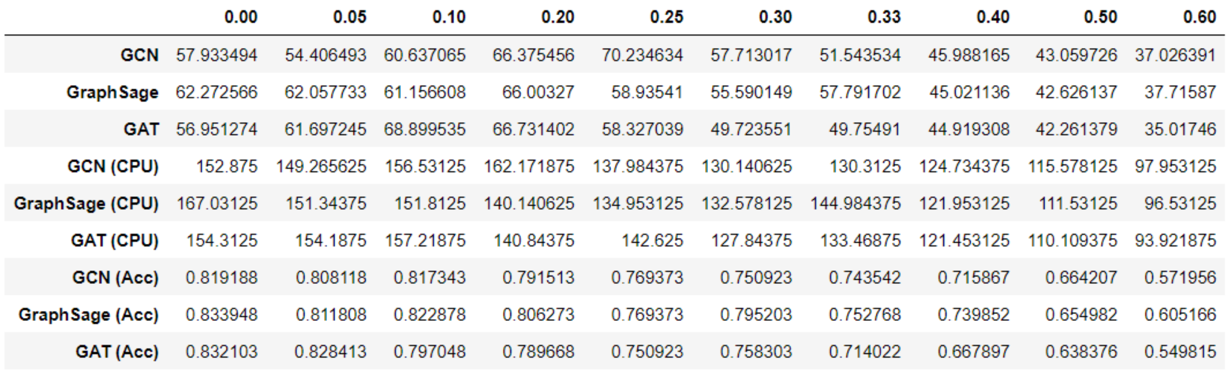
- The below table holds the computational times, cpu computation times and accuracies for different sparsification % across different GNN models.



	0.00	0.05	0.10	0.20	0.25	0.30	0.33	0.40	0.50	0.60
GCN	58.394857	71.052935	64.884055	58.206689	59.188978	54.794622	46.853617	47.454783	44.382234	37.936651
GraphSage	62.108561	59.576123	56.792769	59.149919	54.56934	52.321896	48.942352	48.964971	50.206527	40.752364
GAT	61.868357	72.868916	63.14756	64.048786	53.592726	50.202626	52.486172	48.029693	45.525408	45.838346
GCN (CPU)	144.6875	158.171875	153.375	142.671875	152.203125	146.03125	128.34375	127.28125	115.578125	101.015625
GraphSage (CPU)	153.453125	154.71875	139.6875	146.984375	149.84375	141.890625	129.21875	127.125	118.78125	106.890625
GAT (CPU)	149.96875	156.28125	147.890625	144.828125	144.984375	135.5625	141.078125	127.296875	112.375	109.09375
GCN (Acc)	0.824723	0.826568	0.782288	0.769373	0.776753	0.761993	0.706642	0.688192	0.649446	0.584871
GraphSage (Acc)	0.822878	0.793358	0.800738	0.776753	0.756458	0.736162	0.734317	0.715867	0.658672	0.583026
GAT (Acc)	0.846863	0.791513	0.789668	0.791513	0.763838	0.749077	0.741697	0.710332	0.669742	0.603321

Random-Link Prediction Heuristic (Randomly choose edges to delete based on the Link Prediction probabilities) -

- Initially all GNN models' accuracies vary and are better than the others but the GraphSage model performs better than the others.
- There is an initial rise in time even with 20-30% sparsification which is unexpected but could be attributed to other factors (system computation).
- The below table holds the computational times, cpu computation times and accuracies for different sparsification % across different GNN models.



	0.00	0.05	0.10	0.20	0.25	0.30	0.33	0.40	0.50	0.60
GCN	57.933494	54.406493	60.637065	66.375456	70.234634	57.713017	51.543534	45.988165	43.059726	37.026391
GraphSage	62.272566	62.057733	61.156608	66.00327	58.93541	55.590149	57.791702	45.021136	42.626137	37.71587
GAT	56.951274	61.697245	68.899535	66.731402	58.327039	49.723551	49.75491	44.919308	42.261379	35.01746
GCN (CPU)	152.875	149.265625	156.53125	162.171875	137.984375	130.140625	130.3125	124.734375	115.578125	97.953125
GraphSage (CPU)	167.03125	151.34375	151.8125	140.140625	134.953125	132.578125	144.984375	121.953125	111.53125	96.53125
GAT (CPU)	154.3125	154.1875	157.21875	140.84375	142.625	127.84375	133.46875	121.453125	110.109375	93.921875
GCN (Acc)	0.819188	0.808118	0.817343	0.791513	0.769373	0.750923	0.743542	0.715867	0.664207	0.571956
GraphSage (Acc)	0.833948	0.811808	0.822878	0.806273	0.769373	0.795203	0.752768	0.739852	0.654982	0.605166
GAT (Acc)	0.832103	0.828413	0.797048	0.789668	0.750923	0.758303	0.714022	0.667897	0.638376	0.549815

The hyperparameters like the learning rate, number of layers, weight decay, number of trees were chosen after repeatedly testing with different values and coming to the best possible value based on the accuracy achieved.

- Learning Rate - 0.01
- Number of layers - 2
- Weight decay - 5e-4
- Number of trees (Random Forest) - 100

CONCLUSIONS

Key Findings -

- Based on the above results, it can be concluded that both heuristics have very similar results and no heuristic is better than the other by a significant margin.
- The computational times have significantly decreased as expected.

Limitations -

- Our approach is not designed to handle weighted edges or directed graphs.
- Assessment of the effectiveness of our heuristic on different network types to determine its suitability and reproducibility.

FUTURE WORK

1. Utilize other datasets like PubMed dataset (Planetoid dataset) and understand the effect of our heuristic on the accuracy, computational time and sustainability.
2. Incorporate weighted edges in the heuristic and analyze the impact on the different directed networks.
3. Work on a varying number of nodes and edges in the graph - small to very large graphs to understand the impact on their computational times.
4. Perform a comparative study of sparsification methods with the most recent ones and existing known techniques such as Cut sparsification, Spectral sparsification etc while contrasting their performance on GNNs.

TEAM MEMBER CONTRIBUTIONS

Deepal: Worked on creating a function for random heuristic. Helped Rahul and brainstormed with him in the first and second phase of the project. Created the project presentation. Helped in creating the report. Helped Tarun in the creation of link prediction functions.

Tarun: Worked on creating GCN for running a dataset and then understanding the accuracy of GCN for that dataset. Freeze the last layer and get the node embeddings. Once node embeddings are ready, using that as input to the link prediction function, where this function has been deployed in multiple ways to compare different algorithms and also worked on checking overfitting or underfitting of data. Also employed methods like dropouts , regularization and changing the number of estimators for Random Forest tree for the same. Also helped in creating the report. Helped in project presentation

Rahul: I initiated the proposal to utilize link prediction as a heuristic approach for biased sparsification in the project. Additionally, I developed and implemented both the link prediction and sparsification functions, which were utilized in the project's first phase. I effectively linked the outcomes from the first phase with the second and third phases and performed thorough analysis on the results.

Raghav: I worked with Rahul and assisted him through brainstorming the approach and also in developing the heuristic approach. I also repeatedly tested the code to handle any exceptions which did occur. Moreover, I generalized the code to take in any kind of parameter for code testing, achieving reproducible results with minimal changes and making the code more coherent. Additionally, I helped in creating the report.

REFERENCES:

1) Bojchevski, A. and Günnemann, S., 2017. Deep gaussian embedding of graphs: Unsupervised inductive learning via ranking. *arXiv preprint arXiv:1707.03815*.
2) Berg, R.V.D., Kipf, T.N. and Welling, M., 2017. Graph convolutional matrix completion. *arXiv preprint arXiv:1706.02263*.
3) Ying, R., He, R., Chen, K., Eksombatchai, P., Hamilton, W.L. and Leskovec, J., 2018, July. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining* (pp. 974-983).
4) M. Wu, Y. Huang, L. Zhao and Y. He, "Link Prediction Based on Random Forest in Signed Social Networks," 2018 10th International Conference on Intelligent Human-Machine Systems and Cybernetics (IHMSC), Hangzhou, China, 2018, pp. 251-256, doi: 10.1109/IHMSC.2018.10164.
5) Co, Jan Miles & Fernandez, Proceso. (2016). Link Prediction in a Weighted Network using Support Vector Machine. 10.18178/wcse.2016.06.083.