Studying Explain-ability and Comparing Graph Neural Network models for Link prediction task

Abdullah Khan, Sahely Bhadra

Abstract

Link prediction has been an area of extensive research for applications built over graph-based databases. The problem of link prediction is not so new and several algorithms have been studied in this direction. However, the use of Graph Neural Networks is a novel technique that makes these solutions much more generalizable and at the same time make them more scalable. Given that the use of GNNs to encode the nodes of a graph is based not only on its features but also the topology of the graph itself, we are able to leverage the already known autoencoder framework to predict missing links. We compare various graph neural network architectures, to determine the affect they might as encoders for link prediction task. Node embeddings (Hamilton, 2017) are extremely useful as features for link prediction, where the goal is to predict missing edges, or edges that are likely to form in future.

Introduction

Link prediction has been an area of extensive research for applications built over graph-based databases. While primarily link prediction lies at the core of most recommender system (J. Ben Schafer, 2007), notable implementations include social network-based relationship prediction (Wang, 2015) (suggested friends, people you may want to follow etc.); biomedical implementations (drug-protein (Stanfield, 2017), protein-protein interactions (Yanjun Qi, 2006) etc.). Various methods for link prediction have been studied, these include heuristic methods, embedding methods, and feature-based methods. Heuristic methods compute some heuristic node similarity scores as the likelihood of links (Liben-Nowell, 2007), such as common neighbors, preferential attachment (Barabasi, 1999), and Katz index (Katz, 1953). Recently GNNs have emerged as powerful tools for representation learning on various downstream machine learning tasks (Hamilton, 2017) including node classification, graph classification and link prediction. Various GNN architectures have been proposed which use a variation of such techniques to find node embeddings. We evaluate the effect of such models on link prediction task; the models used are

GCN (Kipf, 2016), GAT (Veličković, 2017)(attention mechanism in GNNs) and GraphSAGE (Hamilton, 2017) (inductive learning). Using an autoencoder framework (Kipf, 2016) we compare these different models, while also comparing GAE and VGAE (Kipf, 2016) variations of the framework. With the advances of deep learning, current link prediction methods commonly compute features of the nodes of the complete graph. Such pairs of embeddings are then mapped to a singular value assigned to that link. After such a transformation, the link prediction task reduces to a classification task with sample points coming from the existing links (positive links) and a sample of non-existent links (negative links). The transformations that map a pair of embeddings to a single embedding, vary with most commonly used technique being Inner Product, other decoding techniques based on tensor factorization also exists such as DEDICOM (Papalexakis, 2017)and RESCAL (Papalexakis, 2017).

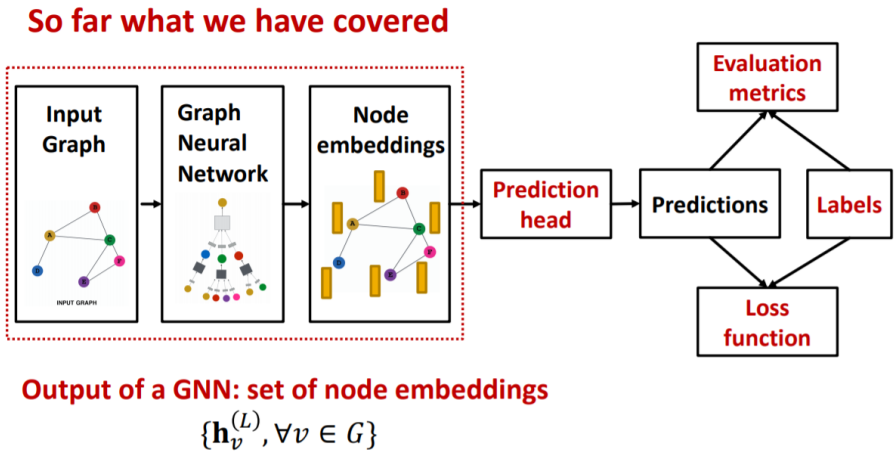
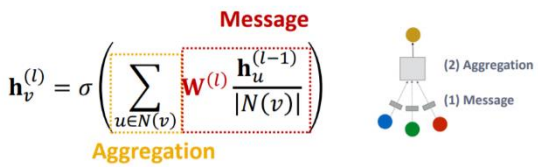


Figure 1: shows a general GNN pipeline. The loss function can be used for controlling both the node embeddings as well as prediction head generation

Our first experiment is to see the effects of using different GNN architectures for generating these node embeddings. We will compare some of the most common GNN architectures used which differ from each other not only in their way of aggregation of surrounding embeddings, but may also choose to operate in an inductive manner. The idea behind any graph neural network is that the computation graph formed by the neighbors of a node represent the topology around it and are influence in the properties of that node. Thus, we want to convolve the information of the surrounding neighbors of a node captured by its computation graph, and aggregate such information in order to encode the node such that similar nodes end up near each other in the high-level embedding space. This process involves generation of node embeddings (**Message Passing**) based on local network neighbors and then aggregating (**Aggregation**) it using a suitable function. Thus, a GNN model is completely defined by its Message Passing and aggregation scheme.



Following is the MP + A schemes of various models being compared as our experiment.

Figure 2: shows the GCN layer. $h_v^{(l)}$ represents the embedding of node v for the layer l. $h_u^{(l-1)}$ represents the embeddings of the nodes on the neighborhood of v calculated in the previous layer.

GCN uses a normalized message passing scheme, which is normalized by the node degree. While its aggregation function is simple summation of all these messages. GraphSAGE uses a generalized aggregation scheme which is taken as an input from the user and also uses self-embedding for final aggregation. This user input may be any aggregation scheme such as MEAN, Pool, or LSTM, or any other custom defined scheme.

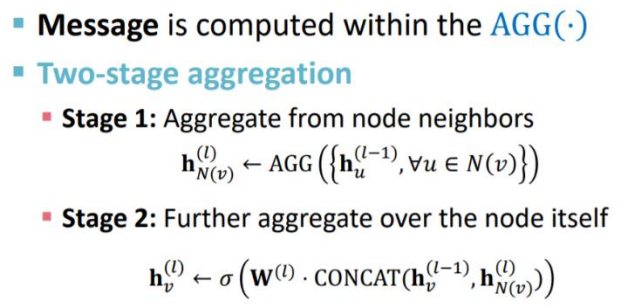


Figure 3: GraphSAGE Message Passing and Aggregation scheme. Aggregation happens in two passes, and is concatenated afterwards. L2 normalization may be applied after every layer in case embedding vectors have different scales.

GAT adds the idea of attention weights to the aggregation schemes, by assigning different learnable importance factor (α_{uv}). GATs also allow for various aggregation functions, but the most commonly used one is SUM and thus, we will also use the same in our experiments.

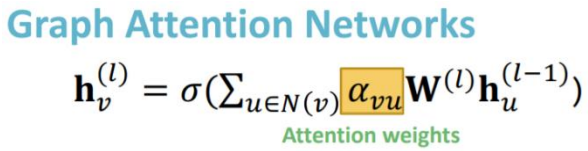


Figure 4: shows a general GAT layer. The attention weights are parametrized and many modern GAT implementations use multi-head attendtions.

Current Work

Our current implementation involves splitting the dataset into train and test edges. We need to hide some of the edges from the GCN and let it predict if those edges exist. We divide the given edges in a graph into two sets, *Message Edges* and *Supervision Edges* where message edges are used in message passing of the GCNN algorithm, while supervision edges are used for supervision of edge predictions made by the GNN model while in the training phase. They are kept away from the model, and are further split into train / test / val edges which make the datapoints of the classifier. We use a **transductive** (Hamilton, 2017) approach in this dataset split, i.e., the entire graph topology is fed to the encoding model as opposed to the **inductive** (Hamilton, 2017)

approach which feeds separate graphs as train and test sets. This process is presented in a concise manner by Figure 5.

Negative Samples

Now, we have got a well split dataset with links that actually exist. However, at the very least we are training a binary classifier to predict the edges from the non-edges (the negative samples of our datapoints). We do so by taking one end point from the present link and match it with another node such that the link does not exists in our original graph in order to produce a negative sample for our classifier. We generate same number of negative links as there as positive in order to remove the hassle of dealing with skewed data.

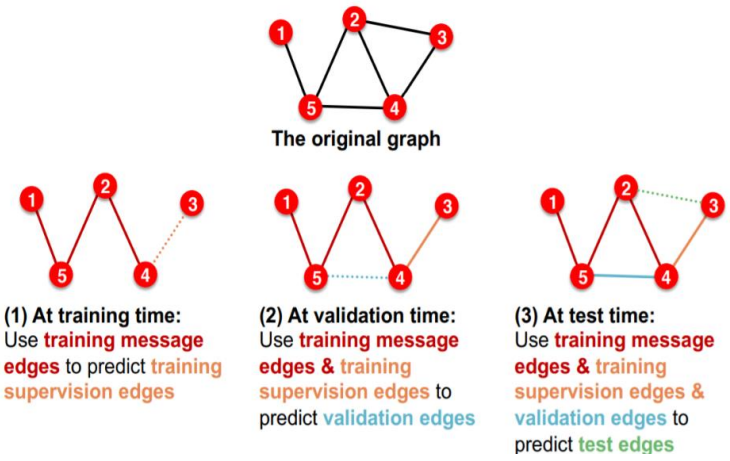


Figure 5: The dataset is split in a total of 4 types of edges:

1. training message
2. training supervision
3. validation
4. test.

The number of edges known to the GNN model keeps on increasing as the supervision edges become known to the model after the training stage.

Auto-encoder

The traditional autoencoder (Doersch, 2016) is a neural network that contains an encoder and a decoder. The encoder takes a data point X as input and converts it to a lower-dimensional representation (embedding) Z. The decoder takes the lower-dimensional representation Z and returns a reconstruction of the original input X-hat that looks like the input X.

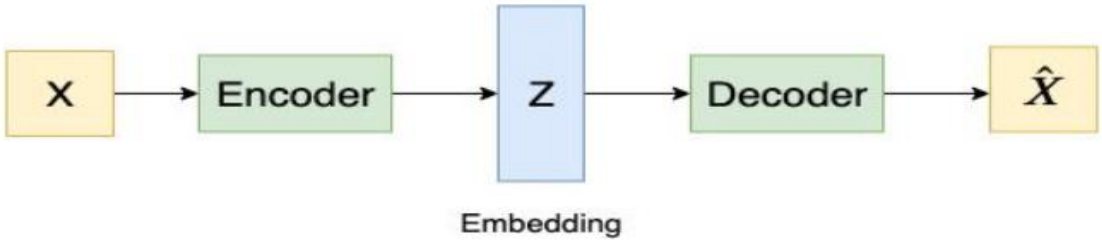


Figure 6: A general autoencoder framework

The main idea of a variational autoencoder is that it embeds the input X to a distribution rather than a point. And then a random sample Z is taken from the distribution rather than generated from encoder directly.

Decoders

Decoders are responsible for mapping a pair node embedding obtained from the Graph Neural Network to real number denoting the probability of existence of a link between the two nodes. The most common form of decoder used for link prediction is a **Non-parametric** Inner product **decoder**. However, they form the simplest type of decoders which are responsible for identifying the link between a pair of nodes in accordance to the closeness of the two vectors representing those nodes in the latent space. The next type of decoder that we introduce are **parametric decoders**. As the name suggest these decoders are able to learn weights/parameters while being optimized along side the GNN embeddings, thus forming an end-to-end training loop. These parametric decoders find their inspiration from tensor factorization techniques commonly used in recommendations, entity matching etc., such as DEDICOM (Papalexakis, 2017) and RESCAL (Nickel, 2011). However, one key difference between those techniques and the one presented here is that we train our model in an end-to-end fashion i.e., GNN embeddings are also getting trained at the same time as the parameters of these decoders. The following table summarizes all the decoders.

Type	**Decoder**	**Mapping function**
Non-Parameterized	Inner Product	$z_i^T z_j$
Parameterized	RESCAL	$z_i^T M_r z_j$
	DEDICOM	$z_i^T D_r R D_r z_j$

Here $\mathbf{z_i}$ is the embedding of the source node while $\mathbf{z_j}$ represents the embedding of the destination node obtained from GNN. In case of RESCAL, we introduce a parameter matrix(M) of size($d * d$)

This matrix is responsible for learning the decoding function and is a generalized weighted matrix for each relation type. In DEDICOM decoder, we introduce a local variant parameter (D_r) and a global variant parameter (R) which are responsible for capturing both the local behavior as well as global behavior of these graphs. The local variant (D) is a diagonal matrix and is later used to find the overall importance of node features across the complete graph. We will discuss on the explanation part in later section.

Experiments

We take the Cora Dataset (Cora, n.d.) for our preliminary testing which is consists of 2708 scientific publications classified into one of seven classes. The citation network consists 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. The dictionary consists of 1433 unique words.

To analyze the structural properties of of our datasets, we compare following network characteristics:

- Degree Distribution
- Hop distribution

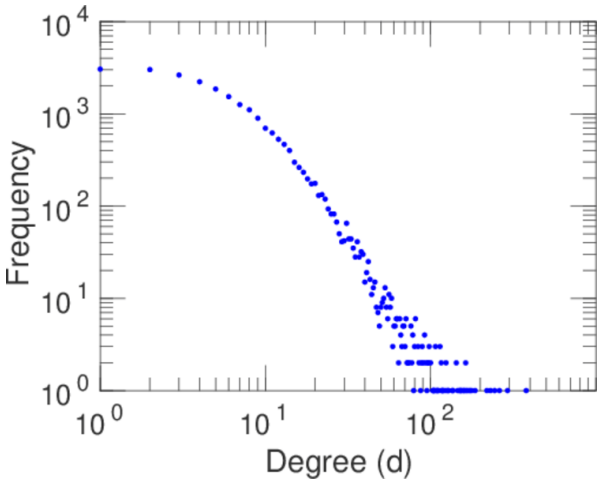


Figure 7: Degree Distribution of Cora Dataset

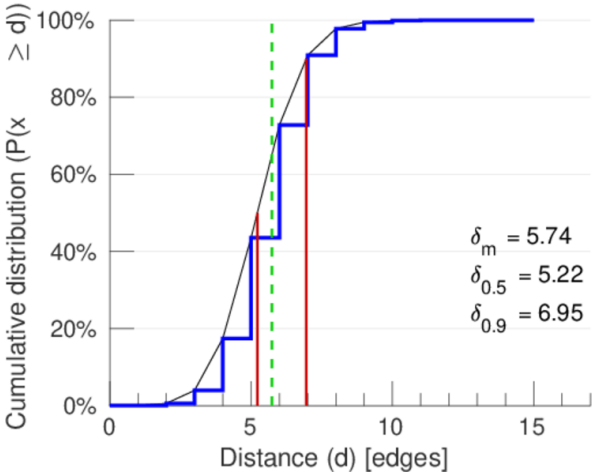


Figure 8: Hop Distribution of Cora Dataset

We also analyze our results on the PubMed Dataset, which consists of 19717 scientific publications from PubMed database pertaining to diabetes 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 which consists of 500 unique words.

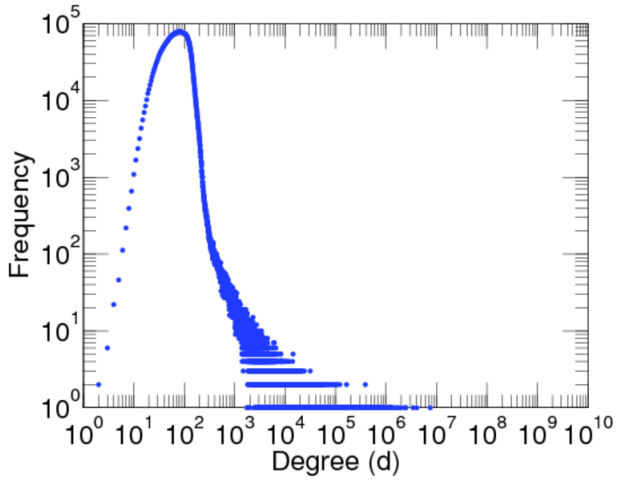


Figure 9: Degree Distribution of PubMed Dataset

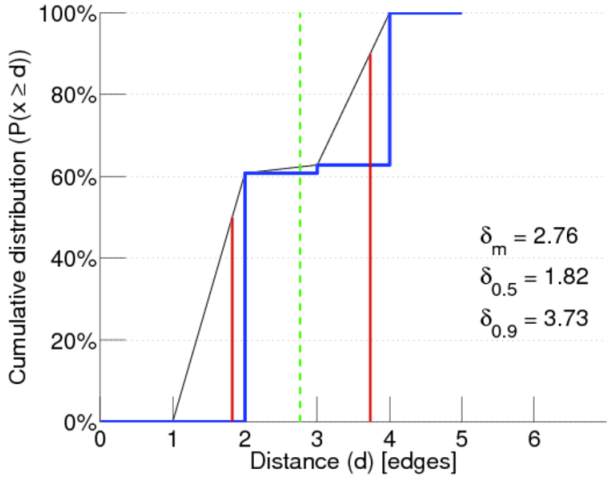


Figure 10: Hop Distribution of PubMed Dataset

From the above graphs, we can see that the PubMed dataset is not only larger but is also denser. The mean distance of the graph is 2.76 edges. Whereas, the Cora dataset is sparser with a mean distance of 5.74 edges. We have chosen these datasets to represent the two broad types of graph datasets found in real world – sparse graphs and dense graphs.

The dataset is split in the ratio of $0.55 : 0.15 : 0.3 \equiv$ train : val : test. We run the model for 200 epochs and evaluate its performance on the maximum test set value achieved by the model for the metrics: AUCROC and average precision (AP). We also measure the recall rate on the final test set as well as produce a confusion matrix for each of the models.

Metrics

AUC ROC: captures the tradeoff in TPR and FPR as the classification threshold is varied for a binary classifier. It gives us the intuition of the probability that a classifier will rank a randomly chosen positive instance higher than a randomly chosen negative one.

Confusion matrix

	Actually Positive (1)	Actually Negative (0)
Predicted Positive (1)	True Positives (TPs)	False Positives (FPs)
Predicted Negative (0)	False Negatives (FNs)	True Negatives (TNs)

Precision (P):

$$\frac{TP}{TP + FP}$$

Recall (R):

$$\frac{TP}{TP + FN}$$

the number of correct positive predictions made. Precision, he minority class.

Recall: gives us a performance indication of the model on predicting actual links correctly. It quantifies the number of correct positive predictions made out of all positive predictions that could have been made. Unlike precision that only comments on the correct positive predictions out of all positive predictions, recall provides an indication of missed positive predictions.

Maximizing precision will minimize the number false positives, whereas maximizing the recall will minimize the number of false negatives

We run tests over both datasets, for all the models as well as the two frameworks; GAE & VGAE. The above stated metrics are calculated and averaged over 10 runs with the following hyperparameters:

Trainer		
Lr: 0.01	Epochs: 200	Weight decay: 0.0005
Models: GCN, GAT, GraphSAGE		
Num_layers: 2	Hidden_channels: 16	Droprate: 0.2

Table 1: Hyperparameters

Results

Table 2 shows the results obtained by performing the experiment on the Cora Dataset. The scores of the three models are pretty similar, thus indicating that the choice of model does not make much difference. The models find different embeddings however, the decoder combines the embeddings and the whole end-to-end system is optimized by the Binary Cross Entropy Classification loss. The difference between VGAE and GAE scores are less than 0.03 % of each other with either performing better in some cases. Thus, only GAE scores have been shown, as it is usually the more commonly used framework.

Model	AUC ROC	AP	Recall
GCN	88.03 ± 0.02	87.85 ± 0.00	61.15 ± 0.02
GAT	87.98 ± 0.00	88.13 ± 0.00	68.27 ± 0.03
GraphSAGE	90.39 ± 0.01	92.12 ± 0.01	82.82 ± 0.01

Table 2: Metrics across various models for Cora Dataset.

Table 3 shows the results obtained by performing the experiment on the PubMed Dataset. The scores of the three models are vary quite a lot in case of GCN and we see a detrimental effect of using other models. However, the point to note is that recall scores point a different picture, with GCN suffering a disadvantage. Thus, we cannot rely on a single metric, neither a combination. We should choose a metric which is suited for the final task, giving a high accuracy for the solutions which gives desired results. Again, the difference between VGAE and GAE scores are less than 0.01 % of each other with either performing better in some cases; thus, only GAE scores have been reported.

Model	AUC ROC	AP	Recall
GCN	90.63 ± 0.00	87.68 ± 0.01	45.05 ± 0.01
GAT	82.98 ± 0.01	80.39 ± 0.00	47.12 ± 0.01
GraphSAGE	84.06 ± 0.01	83.66 ± 0.01	50.80 ± 0.00

Table 3: Metrics across various models for PubMed Dataset

Thus, we can conclude from the above observations that not all models perform equally well in all cases. In case of sparse graphs such as Cora, the models performed similarly well. However, when the graph is dense, like the PubMed dataset, the computational graph of every node also becomes more expensive to compute, thus the simple GCN architecture gains an advantage, while more compute intensive models such as GAT and GraphSAGE remains at a disadvantage. However, the embeddings of GCN with higher AUC ROC scores in second dataset were consistently performing poor in case of classifying positive links, which implies that its higher AUC-ROC scores are due to its ability to correctly classify negative edges. Thus, in case of dense graph, not only do various architectures differ significantly, we also have to realize our evaluation based on the downstream task. A fraud detection system will prefer to classify the positive edges correctly in order to find the outlier among them, while a recommendation system, will focus on classification of unseen edges.

Decoders

For testing the variations caused due the various decoders mentioned before, we ran similar tests as before and evaluating auroc score while using GCN as well as GAT as encoders on the Cora dataset. We evaluated their performance by averaging the result over 10 independent runs, while keeping the latent dimension of our auto-encoder model as d = 512. While keeping rest of the hyperparameters remained same, we obtained the following results.

Encoder	Decoder	AUC ROC
GCN	**Inner Product**	**88.03**
	RESCAL	85.54
	DEDICOM	84.10
GAT	Inner Product	87.98
	RESCAL	**88.36**
	DEDICOM	85.71

From the above observation we can see that the parameterized decoders perform slightly worse in such simple graph benchmark such as Cora, which leads us to realize that a trivial implementation of these decoders are not appreciative and they can only take advantage of the parametrization in specific situation (Marinka Zitnik, 2018). A Point to note is that GAT provides more information for the decoders to take advantage of by assigning weights to the message passing edges and we see a relatively similar performance from the parametrized decoders in such cases.

Explanations

The need for explanations may arise in case of link prediction due to the nature of the problem itself. We have a graph which provides us the topology used by GNNs as well as node features in most cases. However, for any supervised machine learning task we also require a label, which is true for a positive link, as for negative samples we take the same missing links which we will later predict upon. We are not certain whether these negative samples are truly negative or just missing links in the incomplete graph. This uncertainty inculcates a need for human intervention while applying such link prediction methods in real life. As mentioned before a typical example of the use case of such a method is in anomaly detection, however in real world situation we still are not

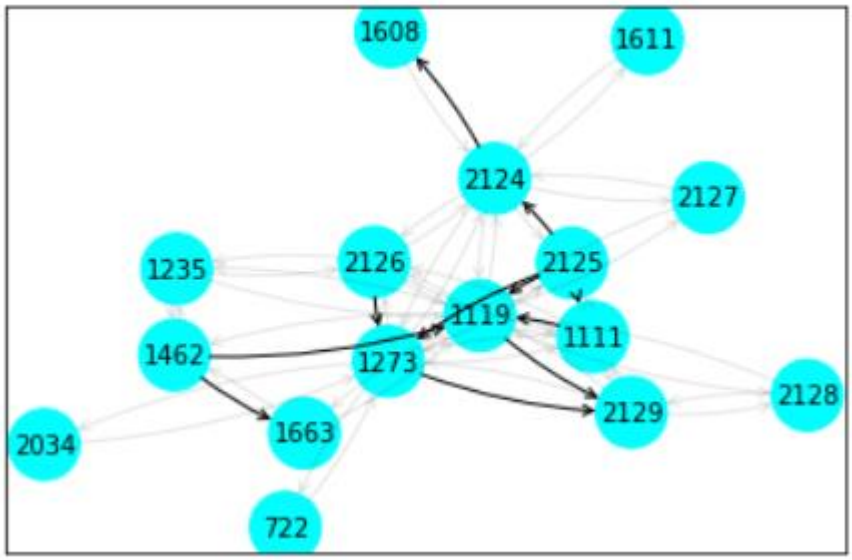
confident upon the machine learning algorithms and thus providing a human interpretable explanation for our predictions eases this uncertainty.

For explaining the decisions made by the GNN model in predicting a certain link, we make use of GNNExplainer (Rex Ying, 2019) which is responsible for identifying the compact subgraph structure along with a subset of node features which are influential in the prediction of a link. We select a 'n' hop subgraph around the interested link and train a mask for both the edges and as well as node features using a classifier which tries to match the weight of the masks based upon the output of the GNN. The importance of a feature or an edge is determined by the mutual information gained by the masks and is formalized as follows:

$$\max_{G} MI(Y,\ G_s, X_s\ ) = H\ Y\ \ - H(Y|G = G_s, X = X_s)$$

Where X_s and G_s are the subset node features and subgraph respectively. H(.) denotes the entropy term.

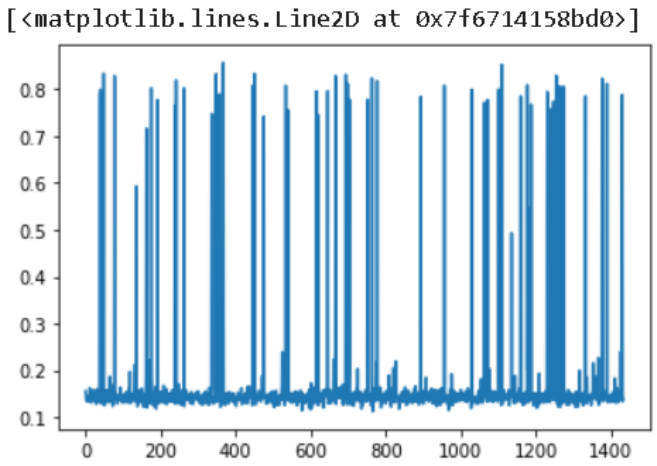
Without going further in detail, what we are interested in is the typical output explanation that we get for a prediction. Typically, we obtain a node feature mask as well as an edge mask which stores the weightage each of these entities have on the prediction. We visualize these explanations by visualizing the compact subgraph obtained from the GNNExplainer using network library. Below is one such example.



We are trying to explain the link between node=1111 and node 2125. We have set our threshold for edge mask to 0.9. The transparent edges are those whose importance falls below the mentioned threshold.

We also take advantage of a commonly used technique in machine learning i.e., **ensemble** to use the output such explanation in order to improve the explainability. The idea behind it is that apart form a subgraph, we calculate the probability of random walks which might lead us from source node to the destination. For this we first calculate all the non-direct paths available from source vertex to destination vertex and calculate the importance of these paths by using the importance of edges obtained from the GNNExplainer and averaging them over the length of the path found in the subgraph. For the above two vertices we obtain the following path which seems to be most **influential**. We confirm our observation by removing such path and finding a decrease in probability of the link from 91% to 78%, when the most influential path i.e. **[1111, 1119, 2129, 2125],** was removed from the graph.

The following is the node feature importance distribution for the **node 1111** as well as **node 2125** combined which was explained before.



As we can see that only a small percentage of nodes features actually account for a large portion of prediction importance.

Future Work

We also tried pruning the node features based on the output of GNNExplainer in order to improve the accuracy of the link prediction task, however, as of now we lack in the experimentation done over such a method as well as an efficient way to calculate the importance of these node features in a scalable manner.

We also tried to come up with the idea of **induced information**, in order to increase the accuracy for the link prediction task by inducing more information in the graph such as relation/edge types which RGCN can take advantage of as well as the parametrized decoders by forcefully assigning

edge types based on the output of GNNExplainer. We tried to do this by calculating the dot-product of the node features and forcefully diving the edges into types only to gain an equivalent result as that of State of the Art.

References

[1] Barabasi, A.-L. a. A. R., 1999. Emergence of scaling in random networks.

[2] Cora, P.C., n.d. *Planetoid Datasets* [Online]Available at:https://pytorch-geometric.readthedocs.io/en/latest/modules/datasets.html#torch_geometric.datasets.Planetoid

[3] Doersch, C., 2016. Tutorial on Variational Autoencoder.

[4] Hamilton, W. L. R. Y. a. J. L., 2017. Inductive representation learning on large graphs.

[5] Hamilton, W. L. Y. R. &. L. J., 2017. Representation learning on graphs: Methods and applications.

[6] J. Ben Schafer, D. F. J. H. a. S. S. 2., 2007. Collaborative filtering recommender systems.

[7] Katz, L., 1953. A new status index derived from sociometric analysis.

[8] Kipf, T. N. &. W. M., 2016. Semi-Supervised Classification with Graph Convolutional Networks. *arXiv preprint arXiv:1609.02907..*

[9] Kipf, T. N. &. W. M., 2016. Variational graph auto-encoders.

[10] Liben-Nowell, D. a. K. J., 2007. The link-prediction problem for social networks.

[11] Nickel, M. T. V. a. K. H.-P., 2011. A three-way model for collective learning on multi-relational data. *ICML, volume 11, pages 809–816.*

[12] Papalexakis, E., 2017. Tensors for data mining and data fusion, models, applications, and scalable algorithms..

[13] Papalexakis, E. E., 2017. Tensors for data mining and data fusion: Models, applications, and scalable algorithms. *ACM TIST*.

[14] Stanfield, Z. C. M. a. K. M., 2017. Drug response prediction as a link prediction problem.

[15] Veličković, P. C. G. C. A. R. A. L. P. &. B. Y., 2017. Graph attention networks.

[16] Wang, P. X. B. W. Y. &. Z. X., 2015. Link prediction in social networks: the state-of-the-art.

[17] Yanjun Qi, Z. B.-J. a. J. K.-S., 2006. Evaluation of different biological data and computational classification methods for use in protein interaction prediction.