Predicting of Protein Interactions using GCNs

Rihab Ziani (Phase 1)
rihabzn@gmail.com

Abderrazzak El Khayari (Phase 2)
Zidanelkhayari@gmail.com

Deepak Rastogi (Phase 3)
rastog01@ads.uni-passau.de

Vishvapalsinhji Ramsinh Parmar (Phase 4)
parmar03@ads.uni-passau.de

Abstract

Proteins play a vital role in repairing tissues, for a block of bones, and blood in the body. Their unique interaction with one another can generate different reactions. It would be interesting to examine and predict various links between them. Using machine learning we tried to predict these links. We discussed and explained **GCN**, how proteins are interacting by analyzing **PPI** datasets, and prepared the data for processing in a **NN** model. Then, we have shown how the NN is built using GCN followed by evaluating the model using different evaluation matrices, where we found the **ROC AUC** score **91.27%** and **AP** score **90.14%** for the test dataset, with specific optimal hyperparameter settings.

1 PROBLEM STATEMENT

1.1 Protein-protein Interactions

Proteins are amino acid chains that ply into a three-dimensional structure that gives them their biochemical function. Proteins use a complex network of interactions with other proteins to perform their function[4]. Prediction of the interaction between two proteins is an important research problem with applications in genetic diseases and pharmacological research that allow us to understand the behavioral processes of life, preventing diseases, and developing new drugs.

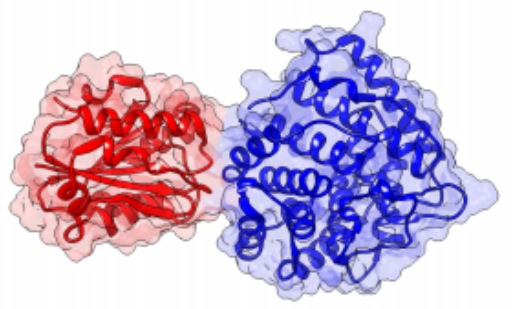

Figure 1: Two proteins in their bound formation[4]

1.2 Introduction to GCNs

Graph convolutional network models receive more and more attention currently in the field of machine learning, they are based on the graph neural network architecture. The concept of GCN is passing a filter through the graph, to search for vertices and edges that can aid to classify the nodes in the graph.

Data Science Lab, 2020, Uni Passau

.

GCN's goal is to encode graph structure and the feature of nodes into low-dimensional representations, and to morph and change these representations to match these node labels.

The idea of GCN is to take graph data which is G=(V,E) where V is a set of vertices and E is a set of edges, and to classify the vertices according to some node label.

A hidden layer in the GCN can be defined as the function[10]:

$$H^{(l+1)} = f(H^{(l)}, A)$$

$H^{(l)}$ is the matrix of activations in the l-th layer, with $H^{(l)} = X$,and X is the feature matrix.

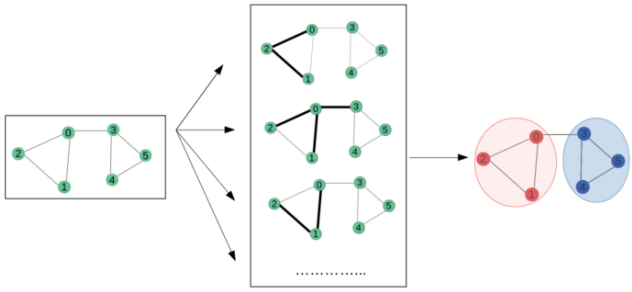

Figure 2: Illustration of Graph Convolutional Networks[1]

Propagation Rule

Every GCN layer identifies a propagation rule that can come in several forms, it sets how inputs will be transformed before being transmitted to the next layer. A simple one is introduced in Kipf Welling [10]:

$$f(H^{(l)}, A) = \sigma(\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}) \quad (1)$$

$\hat{A} = A + I$, where A is the adjacency matrix that define the edges in the graph, and I is the identity matrix that is added to enforce self-loops that are added so that each node includes its own features at the next representations as well as to help with numerical stability.

$\hat{D}$ is the degree matrix of $\hat{A}$ and is used to normalize nodes with large degrees.

The other key idea in this GCN is the symmetric normalization, i.e. $D^{-\frac{1}{2}}AD^{-\frac{1}{2}}$

$\sigma(.)$ denotes a non-linear activation function.

$W^{(l)}$ is a weight matrix for the l-th neural network layer.

[1] https://towardsdatascience.com/understanding-graph-convolutional-networks-for-node-classification-a

Semi-supervised learning

Semi-supervised learning is a task used in a specific type of data to improve the classification performance by training the model of nodes. It is a combination between supervised learning and unsupervised learning techniques, and attempts to tackle the problem of unlabeled data being often plentiful and available, while labeled data being expensive and hard to get.

The semi-supervised learning algorithm for GCNs was introduced in Kipf Welling paper [10] that trains a model by adding some labels and observing the embeddings react. And this is the main method we are using in our project by learning the embedding for each node using the labeled and unlabeled data that we have.

1.3 Formal definition of the problem

There are several computational approaches proposed as complementary to experimental methods for predicting protein-protein interactions. One is the use of Graph Convolutional Networks (GCNs) that will be tackled in our project.

Given two proteins, we predict in our work the probability that they physically interact in a cell, using an operator that generates an embedding for any pair of nodes.

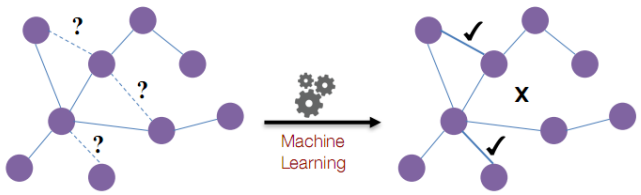

Figure 3: Illustration of prediction protein protein interaction using machine learning [2]

In our work, we use the network to construct a model for predicting new protein-protein interactions by taking the yeast protein-protein interaction network that we have in the data. Then we use a graph convolutional network to solve this prediction problem process that is generated to a link prediction problem.

In link prediction, we have a community of nodes with a certain fraction of edges removed, and our aim is to predict these missing edges.

How to predict missing edges

We start with some basic assumptions about what elements in our data might predict whether two proteins will interact at a later date. the following main approaches are our key steps in increasing the probability of the interaction of two proteins.

- Before training the GCN model to predict the edges, we will present the data as a graph G=(V,E) where V is the set of nodes (proteins) and E is the set of edges (links between proteins), with a network of 6526 nodes and 532180 interactions.
- we split the yeast dataset into training and testing sets, and then for the link prediction, we will first use the GraphConv layer module and we will generate a feature matrix with a dimension 6526x6526.
- We will apply the adjacency matrix A for the model to learn features based on the edges of the nodes.
- For the training, we will use the ReLU activation function and the Sigmoid activation function[3], as the structure of our link prediction model is consisted of two GraphConv layers.
- We will define 4 operators that will be mentioned in the implementation phase (Average Score, Hadamard Product, Weighted-L1, Weighted-L2) to find edge embeddings in the Baseline model we have, and we will evaluate the results we have using the best amongst them.
- In our project, we will train a baseline model, with the same training set for training the GCN model, to compare it with the GCN model in terms of performance, to compare the score of both models.

2 Data Acquisition & Pre-Processing

2.1 Data acquisition

2.1.1 Meaning of data acquisition

Data acquisition is the process for fetching filtering, and cleaning data before the data is stored in any other storage solution. The main goal is to produce a clean and high-quality data set. data acquisition is done before or in parallel while training your model to avoid noisy data sets or missing values.

2.1.2 Source of data

We can consider the data sets for the protein-protein interaction as graph. Here we are referring data set from the yeast edge-list provided by stanford [14]. The data set was displayed as a web page within a HTML tag. We used the requests.get method from requests[4] to fetch data and save it in a yeast.edgelist file.

The protein-protein interaction (figure 4,), is presented as a G=(V,E) where V is the set of nodes(proteins) and E is the set of edges(links between proteins).

2.1.3 PPIs Network Description.

Our yeast protein–protein interactions network consist of 532.180 positive samples, and 42.049.970 of negative samples [18], that means we have an unequal distribution of classes within our dataset. To fix this issue we are going to balance the data by removing edges to get 50% negative data and 50% positive data, how unbiased negative examples are chosen in the under-sampling process with a high performance. This procedure can be done in many ways and we suggest to choose negative link that belongs to the same biological process (the same component).

The heatmap below (figure 5) demonstrates that This dataset should be balanced for representing each class. It should be balanced with 50% instances of each class (positive, negative).

[2] http://snap.stanford.edu/deepnetbio-ismb/slides/deepnetbio-part0-intro.pdf

[3] https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6

[4] https://pypi.org/project/requests/2.7.0/

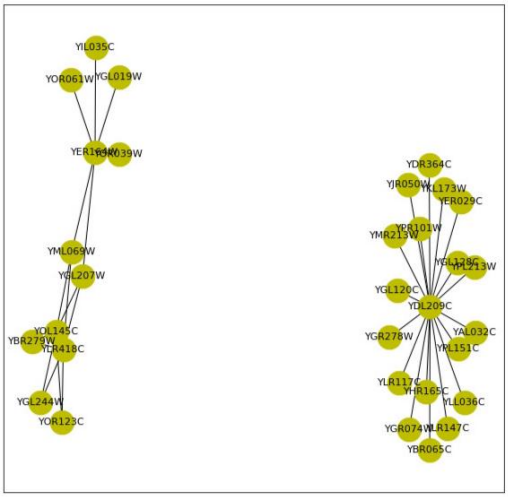

Figure 4: Sub graph of Protein-protein interaction network of yeast (a local part). The nodes are proteins and the links are interactions between them.

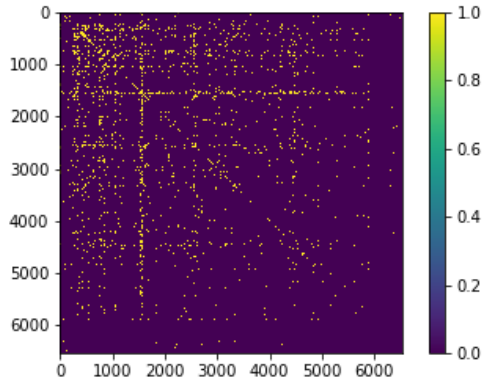

Figure 5: a Heatmap for the Adjacency matrix

2.2 Data preprocessing

Our network consist of 6526 nodes and 532180 interactions, proteins in a specific process should be more likely to interact than proteins in general process, and to interact physically proteins must exist in close proximity near to each other that means co-localization may helps the prediction of unseen links between proteins.

2.2.1 Statistics on the data

*Degree Centrality and essentiality Degree centrality is an algorithm that counts how many neighbors a node has also is used to understand the role of a specific node in a protein–protein interaction network and the importance of that node, where the proteins are the nodes. For example, a protein with high degree is an essential protein. The histogram figure 6, shows the top 10 proteins with high degree, the removal of a central hub protein causes lethality due to the disruption of the interaction.

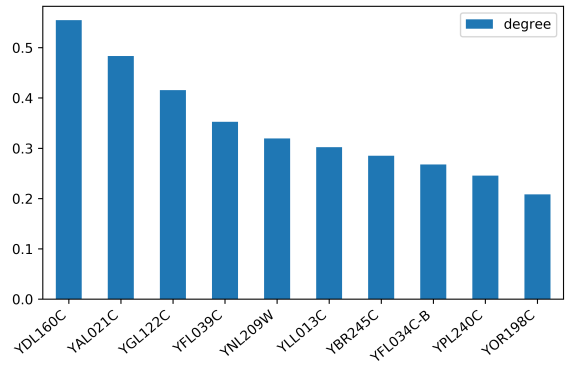

Figure 6: Top 10 essential proteins

The density of PPIs network have a low density with 2.4% that leads to a sparse network.

2.2.2 Removing self-edges strategy

Proteins rarely interact alone. Hence we provide a method to remove self-edges by constructing an adjacency matrix **A** of the undirected graph G=(V,E) where, $A_{i,j}$=1, if (i,j) $\in E$, $A_{i,j}$=0, otherwise after that applies the following operation:

$$\hat{A} = A - I \tag{2}$$

Such that $\hat{A}$ is the new adjacency matrix and I is the identity matrix.

2.2.3 Structural Patterns in PPI Networks

After the analysis of the graph, we use Jaccard Similarity to measure the interface similarity not predicting the link between two proteins. That means two proteins with high Jaccard similarity do not necessarily interact with each other or increase the probability. Assume that the link between YML069W and YLR41bC is missed in figure 7, since we have high similarity between YML069W's neighbor and YLR41bC then YML069W and YLR41bC may interact[3].

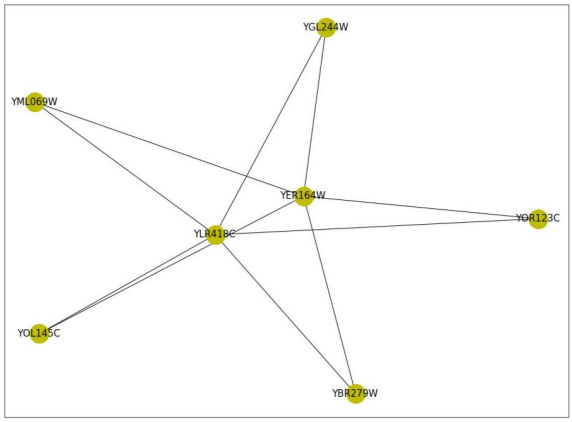

Figure 7: YLR418C, YER164W and their neighbors in the PPI network from yeast dataset

2.2.4 Splitting the data

To split the yeast dataset into training and testing sets and make sure that splitted networks are composed of connected components to avoid the existence of unknown edges.

The dataset is already preprocessed to some extent therefore there is very little to no need of data preprocessing required for our dataset. Although, like any other machine learning task, in order to evaluate the model, splitting of data is done, but in a completely different manner. For splitting the dataset we use a library called Stellargraph which has a utility class named EdgeSplitter which is especially designed to split edges making sure that the resultant graph is complete and no edges and the unknown (or exiting in another set) edges are not considered same as the negative edges. We first prepare our test set. We sample random 25% edges and equal non edges in the test set. The method returns the graph with the selected edges removed, node pairs of the edges and non-edges and their label (0 or 1). Our test consists of 133045 positive and 133045 negative edges. For the training set, Edgesplitter excludes the edges those we have in our test set and randomly similarly selects 25% edges and non edges. Therefore, our training graph has the test edges plus another 25% more edges removed. It has 99783 positive and 99783 negative edges.

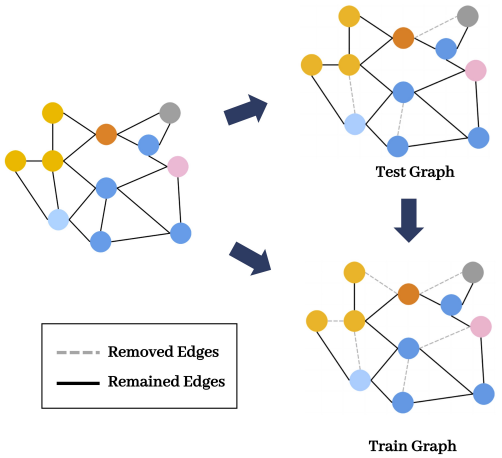

Figure 8: Graph Splitting using EdgeSplitter utility

3 Implementation

The previous section explained how the Yeast network splits into train, validation, and test graph by eliminating a fixed amount of edges in each set. This method also returns an array of the edges removed from that particular graph and an equal number non-edges. In this phase, we discuss how we implement a GCN model that predicts the edges we removed while splitting our graph. We briefly discuss its architecture, training methods, transforming node embedding to edge features, and how the *GraphConv* layer estimates the node features. We also implement a baseline model using Node2Vec to compare its performance with our GCN model.

3.1 Feature Matrix

Generating a feature matrix for the Yeast graph could be a tricky task. We have not been provided any properties of the nodes along with the graph, unlike DGL [15] provides a feature matrix explicitly to embed that information into a feature matrix for this task. Therefore, it is evident that we have to form some features by ourselves. In [10], the author takes an identity matrix as the feature matrix for Karate Club Network. In that case, also, there are no features available for the nodes. Therefore, in our implementation, we assume $X = I$ as the input features. This identity matrix (i.e., feature matrix) has a dimension of 6526 X 6526, and it contains a unique feature for each node.

3.2 Link Prediction Model

For creating the edge prediction model, we use the GraphConv layer module, a predefined graph convolution layer defined in DGL, version 0.5 [15], for learning node representation for our graph. As discussed in [10], A graph neural network model generally learns the representation by taking the graph's structure and the feature representations into account. Similarly, the GraphConv layer works on the principle of the Weisfeiler-Lehman-1 algorithm [16]. This algorithm states a method for assigning node coloring to each node in a graph. The GraphConv layer implements the same hash function used by this algorithm as a neural network like differentiable with training parameters as follows.

$$h_i^{(l+1)} = \sigma(b^{(l)} + \sum_{j \in \mathcal{N}(i)} \frac{1}{c_{ij}} h_j^{(l)} W^{(l)}) \tag{3}$$

Here, $\mathcal{N}(i)$ is the set of neighbours of node i. c_{ij} is obtained using the product of the sqare root of the node degrees $\sqrt{|\mathcal{N}(i)|}\sqrt{|\mathcal{N}(j)|}$. σ is the non linear activation function and $h_i^{(l+1)}$ is the vector of activations of the node i set off by the $(l+1)^{th}$ layer. Putting all these value together in the equation, we achieve the propagation rule, given in equation 1, in vector form. The model parameters $W^{(l)}$ are initialized using Glorot uniform initialization and the bias $b^{(l)}$ is initialized to zero.

The next step in Graph Convolution involves the insertion of adjacency matrix A. The adjacency matrix outlines the edges between the nodes during the forward propagation; consequently, it equips the model to learn features based on the node's edges. The GCN model learns this information using message passing, where the information is propagated among the neighboring nodes within the graph. The Message passing technique is carried out using two functions called the *Message function* and *Update function*. It is defined on the edge and is used to generate the message by combining the information of the edge itself and nodes connected to its ends. The update function is defined on the node to update the node feature by aggregating the incoming message [15].

The GraphConv layer requires two mandatory parameters: the input feature size of $h_j^{(l)}$ and output feature size i.e., number of dimensions of $h_i^{(l+1)}$. Additionally, optional parameters like learnable weights and bias tensors, activation function, and normalizer could be provided. Its forward function accepts the training graph and the feature matrix as the parameters. The layer outputs a feature tensor.

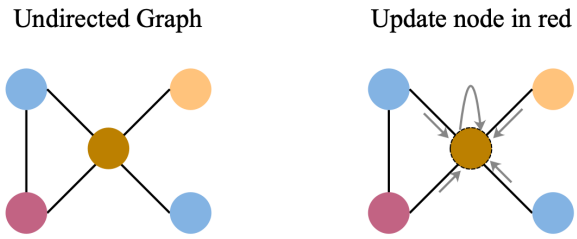

Figure 9: The figure shows the technique of message passing. On the left, we have the undirected graph showing all the edges. On the right, we have the same graph. Grey arrows represent the message m_k being passed to the node in red (center node) where message information is aggregated into the node vector [15].

3.2.1 Model Architecture

Our model's architecture has two main components: first, an ***Encoder*** that is a graph convolution operation operating on the graph ***G*** and producing embeddings for the nodes in ***G*** (thus it is unsupervised). second component is called ***Decoder*** that is simple NN model that predicts Edges after converting node embeddings produced by the *encoder* into edge embeddings.

The *encoder* component of our link prediction model incorporates two GraphConv layers. ReLU activation function is applied to the output tensor of each layer. The decoder component has two layers that are linear layers. The ReLU activation function is applied to the first linear layer's vectors, and the Sigmoid activation function is applied to the output of the second linear layer.

In encoder, the input feature size for GraphConv layer 1 is equal to the total number of nodes in our training graph and the number of output features for this layer, which is also the input size of the GraphConv layer 2 (further, we will call it the hidden layer 1), is varied. The output dimension for the hidden layer 1 is kept fixed i.e., 80. The output tensor from this layer is the final embeddings of the nodes. In decoder, the linear layer 1 (further we will call it hidden layer 2) accepts input of size equal to the output dimension of hidden layer 1, which is 80, and the dimension of the output of this layer is varied. Likewise, the last linear layer accepts the output of the hidden layer 2 as input and outputs into a 1-dimensional tensor. All the values and their influence on variable parameters are discussed in the evaluation section.

3.2.2 Training

For training the model, it needs the training graph $G = (V, E)$, a node representation matrix $X : X \in \mathbb{R}^{(m,n)}$ where m is the number of nodes and n is the number of features for each node (in our case, $m = n$), the edge node pair matrix $P : P \in \mathbb{R}^{(e,2)}$ where e is the total number of edge samples in the training set, and the matrix $y : y \in \mathbb{R}^{(e)}$ having the true link labels (i.e., 0 or 1) of each node pair in P. It is important to note that while training the GCN model, we retain the self-loops in our graph because if there are no self-loops available in the graph G and it performs the dot product of A and X, which sums up the adjacent node features, it does not take its own features into account. So later, when the degree matrix is computed and multiplied with A and X, the product will ultimately be 0, which needs to be avoided [10]. So in the previous section, where the self-loops are removed is only for the data exploration.

Before we begin to train our GCN model, we designate a unique integer to each protein in the graph because every protein as assigned a string label and it would be more apparent to locate node embeddings based on their index (which is also the name of the node) when we estimate edge features. We train this model on Tesla P100-PCIE-16GB GPU with CUDA version 10.1. The two GraphConv layers execute the forward propagation rule given in equation (3), followed by implementing the ReLU activation function after each layer. The second layer produces a vector $emb : emb \in \mathbb{R}^{(m,o)}$ where $o = 80$. emb is the vector representation of the nodes (also called embeddings).

Through encoder, for each node pair P, we obtain the embeddings of each node pair's source and destination nodes by using the node IDs as an index of nodes in the embedding matrix. The feature matrix is constructed so that each node's ID is its index in X. Once the embeddings of source and destination nodes are extracted, a new matrix called $Z : Z \in \mathbb{R}^{(e,o)}$ is computed using an operator f.

$$Z_{(e,o)} = f[emb(P_{e^1}), emb(P_{e^2})] \tag{4}$$

The operator f expects two matrices u and v of identical dimensions, and it results in another matrix Z having the same dimension. In this project, we make use of 4 such operators and estimate our results using the best amongst them. We also use the same operators for determining edge embeddings in our Baseline model. These operators are the following.

1. Average Score is rather simple to calculate. It involves averaging each element of a vector.

$$f_{avg}(u, v) = \frac{u_i + v_i}{2} \tag{5}$$

2. Hadamard Product requires two matrices of identical dimensions and it results in another matrix having the same dimension by performing element-wise multiplication.

$$f_{hdm}(u, v) = (u \circ v)_i = u_i.v_i \tag{6}$$

3. Weighted-L1 is the sum of the absolute difference between the components of the vectors. In this operator, all the components in each vector are weighted equally.

$$f_{L1}(u, v) = |u_i - v_i| \tag{7}$$

4. Weighted-L2 also called Euclidean norm is the squared difference of the components of the vectors.

$$f_{L2}(u, v) = |u_i - v_i|^2 \tag{8}$$

This Z tensor is passed to the hidden layer 2 along with its labels. The output $\hat{y}$ is produced as a probability distribution for i in Z.

The model requires to be trained into batches. Therefore, we make use of utility class provided by pytorch [11] called *Dataloader*. The *DataLoader* works with the instance of the *Dataset* class. Therefore we create a custom class (we call it *EdgeDataset*) that inherits the abstract class *Dataset*. We override two methods, first, *__len__* that returns the size of the dataset and, *__get_item__* to support indexing to access a batch of our dataset using an index i. The *Dataloader* takes the training set node pairs and the link labels and randomly sample the data into the expected number of batches. This sampler can be iterated over to receive our data in batches.

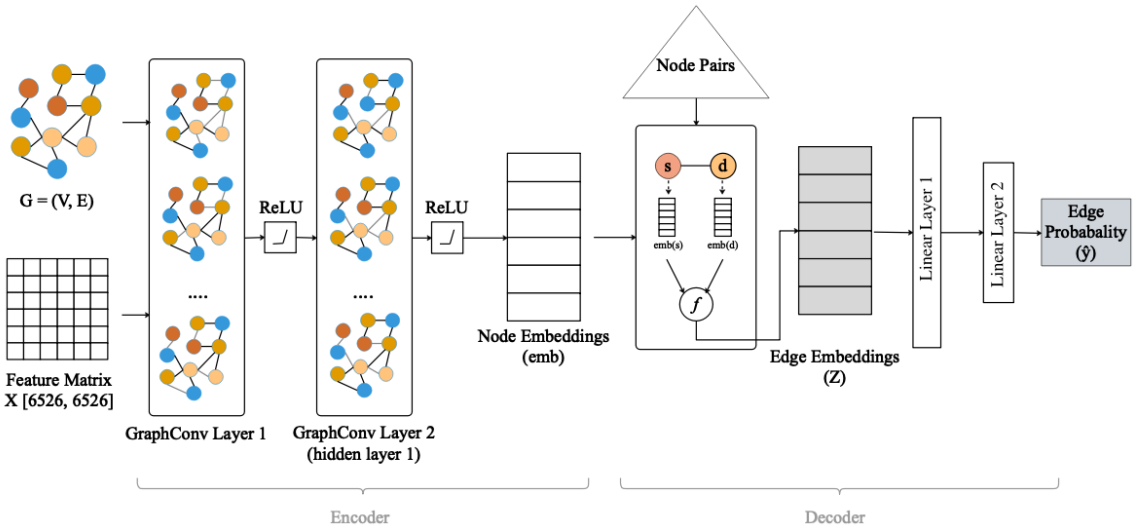

Figure 10: Architecture of Link Prediction Model

3.2.3 Model Optimization

We utilize the **BCE** loss function [11] to measure the loss between y and $\hat{y}$. The function parameters are kept default. For the optimization of the model, we use Adam optimizer. The learning rate is varied. All the values for the learning rate and their influence can be seen in the evaluation section. We calculate the loss and update the weights after training each batch in every epoch.

3.3 Experiments

It could be favorable to have a model with which we can resemble our complex implementation performance. It would not only help us distinguish the scores of two different models but also let us understand what essential aspects of data does a simple conventional model often ignores to consider while a complex model does not. Therefore, we make a baseline model and train it with the same training set for training the GCN model.

Our baseline model has very similar steps as of our GCN model. In the baseline model, we

(1) Learn node embeddings using a well known method called *Node2Vec*.
(2) Convert node embeddings to edge embeddings using the operators discussed in the previous section.
(3) Train a logistic regression model to predict edges.

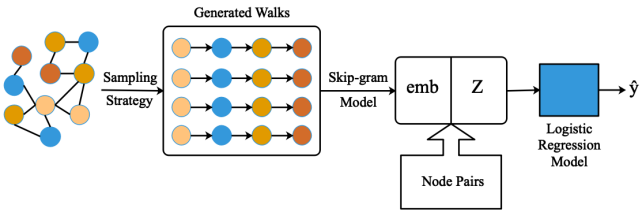

Figure 11: Architecture of the baseline model

3.3.1 Node Embeddings using Node2Vec

The Node2Vec reflects the same strategy as Word2Vec (for learning embeddings for sentences) using a skip-gram model. Since it expects a corpus to be embedded using Word2Vec, Node2Vec needs to follow a similar strategy to embed a graph. However, unlike texts, the graphs could be more complex, such as directed-undirected and cyclic-acyclic. Therefore, to tackle every structure of graphs, it works on a sampling strategy called Random Walks. This sampling strategy requires 4 hyperparameters to be defined [5].

(1) Number of Walks
(2) Walk Length
(3) Return hyperparameter (P)
(4) Inout hyperparameter (Q)

and also the skip-gram parameters (e.g. window size, number of iterations, etc.) The algorithm generates n number of walks from each node with a given walk length. Suppose, while generating a random walk, the algorithms have just transitioned from node t to v, P manages the probability of whether it will go back to the node t or explore other neighbors of v. To determine which nodes the algorithm covers is based on a function that includes the previous node, P and Q, and the edge weights [5].

Once the training of our Node2Vec model concludes, for each node pair P in the edge matrix, where the first node is the source and the second node is the destination of an edge, we use equation 4 to get edge embeddings utilizing the best performing operator from equations 5, 6, 7 and 8. After this step, we have a two-dimensional matrix with edge embeddings ($Z : Z \in \mathbb{R}$), each with dimension 80. We are now able to train a classification model with the true edge labels. Although we evaluate this model with GCN model using the best parameter configurations, we have performed a few experiments seeking the most suitable parameters applying the manual search. We choose to train Node2Vec model with *num_walks* as [10, 18, 24, 30], *walk_length* as [20, 40, 70, 100], P and Q as [0.25,

1] while *window_size* and *Dimension* has been fixed i.e. 10 and 80 respectively.

For this experiment, table 1 shows the set of best-performing parameters. Also, for obtaining edge embeddings, the Average operator (from equation 5) gives the best results.

Table 1: Node2Vec Hyperparameter Values

Hyperparameters	Values
Number of Walks	10
Walk Length	20
P	0.25
Q	0.25
Window Size	10
Dimension	80

3.3.2 Link Prediction using Logistic Regression

To predict the link in our baseline model for the yeast dataset, we use simple and commonly used models like Logistic Regression. Earlier, we discussed how node embeddings are converted to edge embeddings of 80 dimensions, and as true labels, we have a 1 or 0 for if there exists an edge or not, respectively. Therefore, that makes a simple binary classification problem. The logistic regression is a statistical method, very useful for binary classification problems, which computes the probability of an edge existence. The target variable is categorical i.e., 0 or 1. It uses a log of odds as dependent variables and predicts the probability of occurrence of an edge using a logistic function (in this case, Sigmoid).
Let us assume that Z, ($Z \in \mathbb{R}^{(e,o)}$) is our edge feature matrix and t is a linear function of single explanatory variable of Z. t can be expressed as:

$$t = \beta_0 + \sum_{i=1}^{o} \beta_i . Z_i \tag{9}$$

Then general logistic function $\hat{y^1} : \mathbb{R} \rightarrow (0, 1)$ can be written as:

$$\hat{y} = \sigma(t) = \frac{1}{1 + e^{-\beta_0 + \sum_{i=1}^{o} \beta_i . Z_i}} \tag{10}$$

Here $\hat{y}$ is interpreted as the probability of the dependent variable. β_i is the fixed-parameter coefficient, and increasing its value increases the Z_i and eventually increases the log odds. [8]

To train the Logistic Regression model, we create a machine learning pipeline using Sklearn [12] framework. This pipeline consists of two steps that are a *StandardScaler* and *LogisticRegression* Model itself. The idea behind adding a StandardScaler is that it transforms our edge features so that its distribution has a mean value of 0 and a standard deviation value of 1. Since we have around 80 edge features; this is performed for each feature (i.e., column-wise). Here, each value in the dataset has the mean value subtracted and then divided by each feature's standard deviation [12]. This step is done with default parameters.

The LogisticRegression [12] is also trained with default parameters except *max_iter*. A few parameters for example C, which is inverse regularization stength is 1.0 which indicates strong regualrization, *max_iter* is 185 and the *penalty* which specifies the norm used for penalization is *l2* because the default solver algorithm *lbfgs* supports only *l2* norm [12]. The solver typically works as an iterative algorithm that keeps track of weight and bias estimation. It stops either when the algorithm converges (reaches the objective values) or the *max_iter* has been exhausted [12]. Keeping default value for *max_iter*, the algorithm does not converge for our dataset therefore we increase the number of iterations.

4 Evaluation

For our model implementation, it would be helpful to see how effectively it works for the unseen yeast edge list datasets. This process is done by the evaluation which is the next step after the implementation in **KDD** Process[7]. It is done on a dataset, namely training, validation, and testing. The training dataset is used for training our model, the validation dataset is used to find out unbiased results for unseen data, and on the testing, the performance of our model will be evaluated. Evaluation can be understood by answering to the different questions such as how a model is performing for a given data set, which hyperparameter should be given more importance, what kind of performance scores are considered to tell whether a model is performing better or not. These questions are explained in the following sections.

4.1 Commonalities of model

To carry out an evaluation, certain commonalities are being considered. Such as specific splitting of the dataset where training dataset consists of 192371 positive and negative edges, validation dataset includes 6565 positive and negative edges, link embedding generated using operator Hadamard, and fixed hyperparameters as table 2. By altering these commonalities, we are calculating ROC AUC & AP scores which helps to determine relationship between various parameters setting and performance for implemented NN.

Table 2: Default hyperparameters

Hyperparameters	values
Input dim. for hidden layer 1	1024
Input dim. for hidden layer 2	512
Input dim. for output layer	256
Number of epochs	20
Batch size	64
Learning rate	0.001

4.2 ROC AUC & AP scores

Identifying the performance of a model on validation dataset is valuable, it helps to estimate how good our model is at generalization for future unseen data. There are various performance measurement techniques. Two such techniques are used here such as ROC AUC and AP scores. ROC AUC score can find out by calculating the covered area under the ROC curve. The AP score is a summarized form of the precision-recall curve by weighting up the mean of precision function $p(r)$ for recall (r) over each threshold varying from 0 to 1. [1, 19]. Here we are calculating average precision with

the formula,

$$AP = \sum_n (R_n - R_{n-1}) P_n \tag{11}$$

In an equation 11 R_n and P_n is Recall and Precision value of n^{th} threshold [2]. In the following section we are calculating these scores for various changes in default commonalities mentioned in section **??**.

4.3 Stochasticity in learning

It is crucial to know that how our model is performing for constant training-validation dataset ratio. The performance score is observed for the twenty repeats at a fix training-validation ratio. ROC AUC score and AP score is observed for validation set, shown in figure 12 for all the repeats. The standard deviation for the scores is 0.0036 and 0.0043 considering ROC AUC and AP scores respectively which is not high. Thus the effect of stochasticity in the model is neglected.

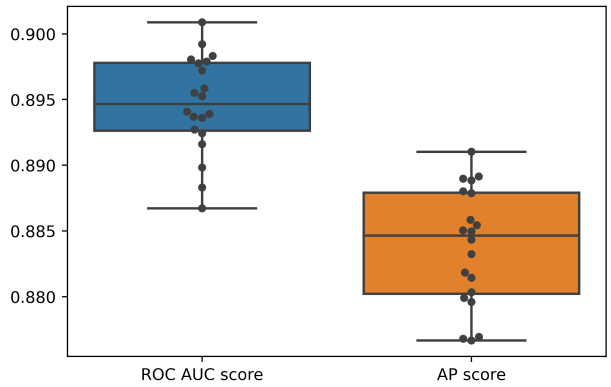

Figure 12: Boxplot of scores for constant train-validation set

4.4 Effect of embedding operator

Embedding for nodes can be generated using different operators such as Hadamard, Average, Weighted L1, and Weighted L2. Scores for the model is fluctuated according to these operators for the validation dataset. These scores can be seen from the table 3. From the table 3 it can be said that the model is performing better for the embedding operator **Average** followed by Hadamard, Weighted L1 and Weighted L2.

Table 3: Scores for a validation set by the different operators

Embedding Operator	ROC AUC score in (%)	AP Score in(%)
Hadamard	89.31	88.35
Average	**89.85**	**88.76**
Weighted L1	89.43	88.30
Weighted L2	89.19	87.90

4.5 Effect of data splitting

Initially, we split the data set into two sets, the test set and the train-validation set. So, for the set of training-validation, we are training our model on a certain percentage(training set) and validate a trained model on a remaining set(validation set). We split the training dataset as shown in a figure 13 with specific percentages. Then we have calculated ROC AUC & AP scores for validation dataset and plot them accordingly. The figure shows that when we train our model on 90% training data, it is giving the highest score among all the splits.

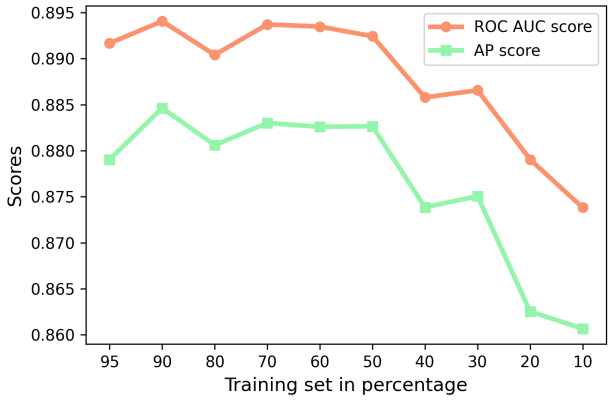

Figure 13: Changes in a score for the validation set by varying the training set

4.6 Hyperparameters of Model

Hyperparameters are the vital knob for training our model, which aids to find out the best performing model. They can be chosen heuristically from certain numbers since there is a no specific formula to choose them. Performance of the model varies according to these hyperparameters. Henceforth, we can change these parameters and find it out that for which of them our model is providing the best performance. The way of optimizing our model by changing these hyperparameters is known as ***hyperparameter tuning***. [9]

For the implemented model we are using hyperparameters such as different sets of units for the hidden layer 1, hidden layer 2, and output layer, number of epochs for which model get trained, the number of batch sizes which specify how many samples should go through before updating the model's internal parameters, learning rate which decides how much changes should be undertaken to update the trainable parameter weight. Units for the input layer are fixed at 6526, the total number of proteins. Values for these hyperparameters are mentioned in table 4.

We choose a different combination of the hyperparameters from table 4 and train and validate our model to find out optimized hyperparameters. On the **validation dataset**, ROC AUC and AP scores are calculated for the changes in respective hyperparameters which are discussed as below.

Table 4: Hyperparameter and value

Hyperparameters	values
Input dim. for hidden layer 1	2048, 1024, 512, 256, 128, 64, 32
Input dim. for hidden layer 2	1024, 512, 256, 128, 64, 32, 16
Input dim. for output layer	512, 256, 128, 64, 32, 16, 8
Number of epochs	[5, 30]
Batch size	32, 64, 128, 256, 512, 1024
Learning rate	0.005, 0.001, 0.01, 0.1

4.6.1 Changes in Epoch

The model performance is changing with the size of epochs. Here we are considering epochs ranging from 5 to 30. From figure 14, it can be said that the model is generating maximum score at epoch 24 for both scores. We have observed scores for further 6 more epoch, but it is not exceeding the scores which obtained at epoch 24. So we can say that global maxima for the model is at 24 for the considered epoch range.

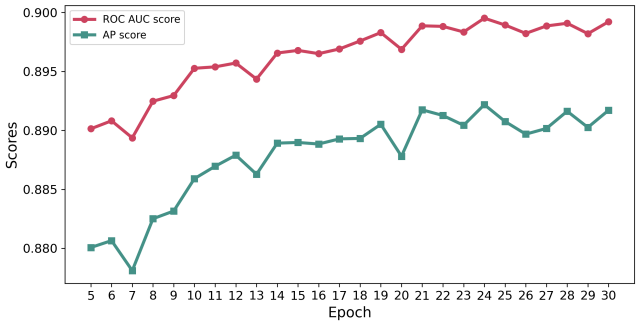

Figure 14: Changes in score with epoch for the validation set

4.6.2 Changes in Batch size

Batch size indicates the total number of samples our model needs to process before updating internal parameters. With variation in these sizes, our model learns differently. Here we are considering six different batch sizes in the power of 2 mentioned in table 4.

According to figure 15 there are some changes in score with the different batch size. From the comparison, it can be said that it gives higher results for batch size 64 while for batch size 32, 512, and 1024 scores are quite low.

4.6.3 Changes in Learning rate

Another change in the score is observed by varying the learning rates, chosen according to table 4. Learning rate is a helpful parameter which decides how much weight to update during the training of NN. Here we are noticing that our model is not learning at all for learning rate 0.1, which gives ROC AUC and AP score of 0.5. Other changes in the scores noticed with different learning rates shown in figure 16. From that, it can deduce that the model is performing better when the learning rate is 0.001.

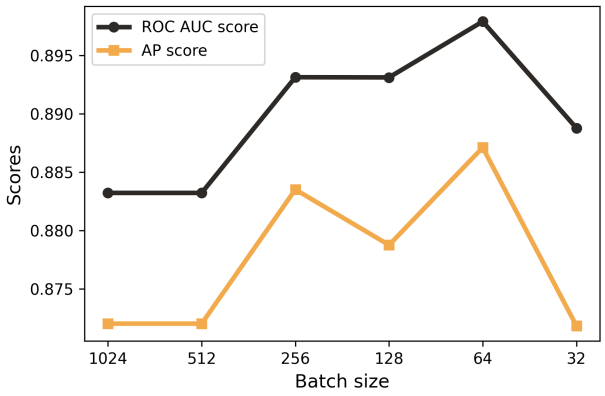

Figure 15: Changes in a score by batch size for the validation set

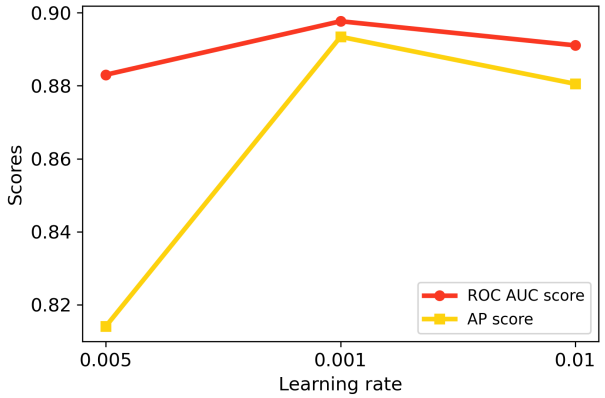

Figure 16: Changes in a score by learning rate for the validation set

4.6.4 Changes in unit size of hidden layer 1

Dimensions of the hidden layer also work as an important factor for training a model. There is a no specific method to choose a unit size of the layers for NN. Here we are changing unit size according to table 4. The effect of these changes depicted in figure 17. From the figure observation, when unit size in a hidden layer was kept at 512 it gives better performance than other settings.

4.6.5 Changes in unit size of hidden layer 2

We are changing unit size for the hidden layer 2 as mention in table 4. The effect of these changes on the scores is reflected in figure 18. When we are decreasing the size from 128 onwards, it is also decreasing performance scores, which does not seem to be beneficial for the model. It shows that, scores for ROC AUC and AP are better when unit size is kept 128.

4.6.6 Changes in unit size of output layer

We are running our model for the seven different unit size at the output layer, mentioned in table 4. The scores are plotted in figure 19. We are getting almost the same high score for a batch size 512

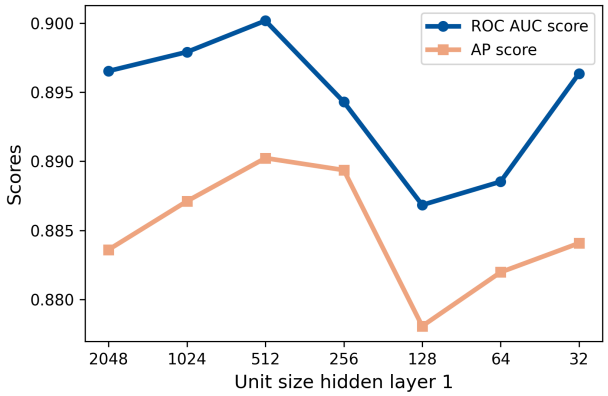

Figure 17: Changes in a score by unit size for the validation set

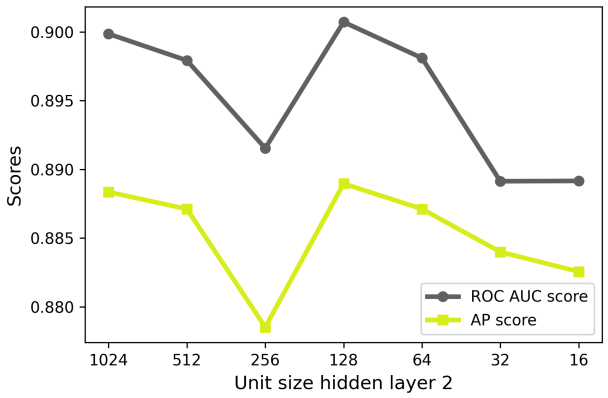

Figure 18: Changes in a score by unit size for the validation set

and 256 when considering ROC AUC score. When evaluating the AP score, it gives a higher score for the unit size 128 followed by 256. It is not beneficial to select batch size 128 since it gives quite a low ROC AUC score. We have recorded the execution time for one epoch which is 306.77 seconds, for the unit size 512, whereas 281.31 seconds for the unit size 256. So, by keeping time factor in mind we are choosing the unit size 256.

4.7 Model Performance

From all the above analysis mentioned in section 4.4, 4.5, and 4.6 we choose embedding operator as average, data split is done where training, validation, and testing dataset consists 179609, 19957, and 266090 positive and negative edges respectively. We have selected hyperparameters as shown in table 5. To find out performance of model for unseen testing data, we are considering evaluation matrices such as CM, ROC curve, ROC AUC score, PR curve, and AP score. These scores are discussed as below.

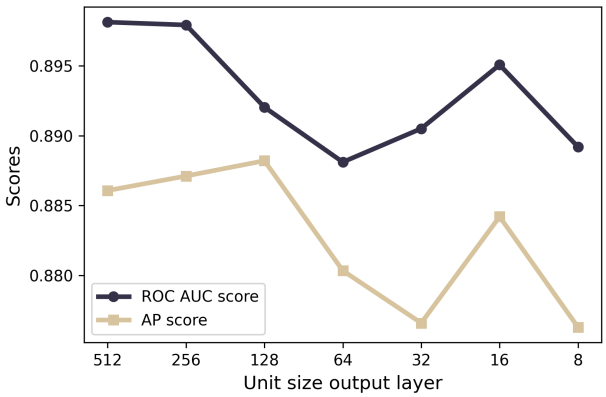

Figure 19: Changes in a score by unit size for the validation set

Table 5: Optimized hyperparameters

Hyperparameters	values
Input dim. for hidden layer 1	512
Input dim. for hidden layer 2	128
Input dim. for output layer	256
Number of epochs	24
Batch size	64
Learning rate	0.001

4.7.1 Confusion Matrix

A confusion matrix is one of the most simple and commonly used matrices in the field of machine learning and data science. It is exceedingly useful despite its simplicity. One can find out interesting results such as precision, recall, F1 score, **FPR**, **TPR**, specificity and many more[17]. The confusion matrix for test dataset can be seen from figure 20

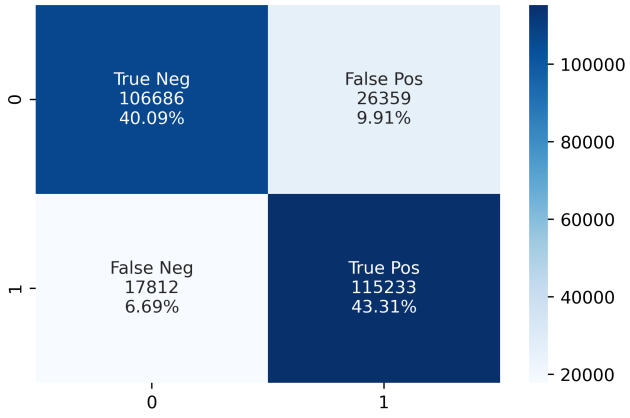

Figure 20: Confusion matrix for test datasets

From figure 20 it can be seen that our model has generated or removed a total 221, 919 edges between nodes correctly. However, it

has predicted or removed total 44, 171 edges between nodes which were there or not at all between nodes. Precision, recall and accuracy scores calculated from this CM is 81.38%, 86.61%, and 83.43% respectively considering a case prediction of links between nodes.

4.7.2 ROC curve

ROC curve finds out the diagnostic ability of a model by varying its threshold. It calculates TPR and FPR for a specific threshold value varying from 0 to 1 and then plot it[6]. TPR and FPR are also known as Sensitivity and 1− Specificity respectively. A perfect model has the area under the ROC curve is 1. Such a model has a 100% separating capacity, assigning a positive and negative point correctly. Here, for the testing data set ROC curve is generated in figure 21. The area under the ROC curve is 0.91 which means, there is a 91% of possibility for a model to assign links between node correctly.

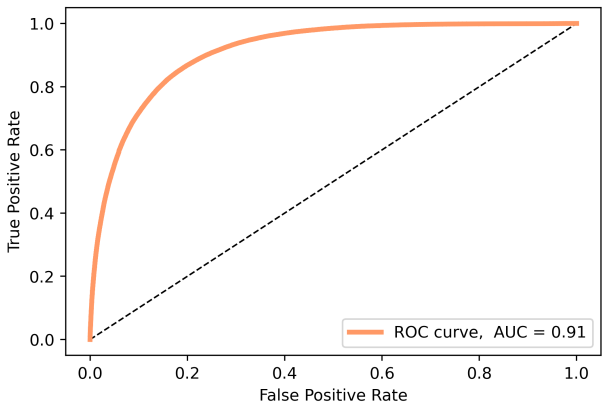

Figure 21: ROC curve for testing datasets

4.7.3 Precision Recall curve

We furthermore implemented the PR curve shown in figure 22. To find out balance within precision and recall by taking different threshold for recall varying between 0 to 1 [13]. From the figure, it can be implied that our model is able to provide a precision of 50% when considering a case of recall with 100%. A higher area under the PR curve suggests a tendency of getting higher the precision and recall score for a model. The total area covered under this is approximately 0.90 which is our AP score.

4.8 Result discussion

After examining the model performance for different hyperparameter combinations from table 4, we come to know that our NN is giving better scores for the hyperparameters and other settings mentioned in the section 4.7. We are achieving better ROC AUC and AP scores for our implemented model created using GCN, which is better than our considered basemodel created using Node2Vec. The ROC AUC and AP scores are mentioned in table 6 for both of them.

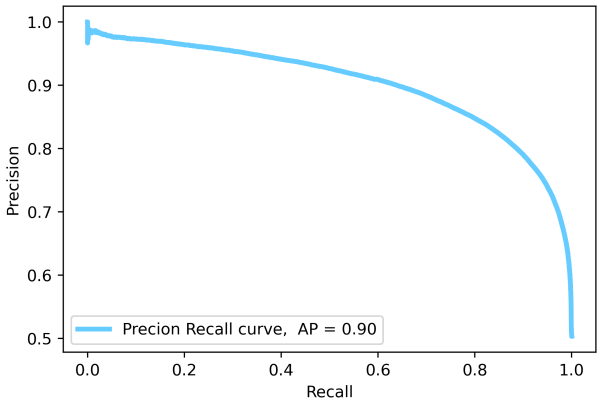

Figure 22: Precision Recall curve for testing dataset

Table 6: Scores of Node2vec and GCN model

Model created using	ROC AUC score in (%)	AP Score in(%)
Node2Vec	70.84	64.34
GCN	**91.27**	**90.14**

5 Acknowledgement

This article was written during the Data Science Lab 2020 at the University of Passau. Our team comprises according to table 7:

Table 7: DSL2020 team members

Phase	Name
I)	Rihab Ziani
II)	Abderrazzak El Khayari
III)	Deepak Rastogi
IV)	Vishvapalsinhji Ramsinh Parmar

Acronyms

AP Average Precision.
AUC Area Under Curve.

BCE Binary Cross Entropy.

CM Confusion Matrix.

DGL Deep Graph Library.

FPR False Positive Rate.

GCN Graph Convolution Network.
GPU Graphics Processing Unit.

KDD Knowledge Discovery Database.

NN Neural Network.

PPI Protein Protein Interaction.
PR Precision Recall.

ROC Receiver Operating Characteristic.
ROC AUC Area Under the ROC Curve.

TPR True Positive Rate.

References

[1] Average precision [n.d.]. *Average Precsion score from wikipedia.* Retrieved July 16, 2020 from https://en.wikipedia.org/w/index.php?title=Information_retrieval&oldid=793358396#Average_precision

[2] Average Precsion score [n.d.]. *Average Precsion score from scikit learn user guide.* Retrieved July 16, 2020 from https://scikit-learn.org/stable/modules/generated/sklearn.metrics.average_precision_score.html

[3] Chen Y, Wang W, Liu J, Feng J, Gong X 2020. *Protein Interface Complementarity and Gene Duplication Improve Link Prediction of Protein-Protein Interaction Network.* Retrieved June 5, 2020 from https://europepmc.org/article/pmc/pmc7142252

[4] Alex Fout, Jonathon Byrd, Basir Shariat, and Asa Ben-Hur. 2017. Protein interface prediction using graph convolutional networks. In *Advances in neural information processing systems.* 6530–6539.

[5] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable Feature Learning for Networks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.*

[6] Karimollah Hajian-Tilaki. 2013. Receiver Operating Characteristic (ROC) Curve Analysis for Medical Diagnostic Test Evaluation. *Caspian journal of internal medicine* 4 (09 2013), 627–635.

[7] J. Han, J. Pei, and M. Kamber. 2011. *Data Mining: Concepts and Techniques.* Elsevier Science. https://books.google.co.in/books?id=pQws07tdpjoC

[8] David W.. Hosmer and Stanley Lemeshow. 2000. *Applied logistic regression.* Wiley New York.

[9] Hyperparameter optimization [n.d.]. *Hyperparameter optimization from wikipedia.* Retrieved July 17, 2020 from https://en.wikipedia.org/wiki/Hyperparameter_optimization

[10] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[11] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. (2017).

[12] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python . *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[13] Precision from scikit learn user guide [n.d.]. *Precision from scikit learn user guide.* Retrieved July 16, 2020 from https://scikit-learn.org/stable/auto_examples/model_selection/plot_precision_recall.html#sphx-glr-auto-examples-model-selection-plot-precision-recall-py

[14] Stanford 2018. *Yeast edgelist - Protin protin Interection.* Retrieved June 5, 2020 from http://snap.stanford.edu/deepnetbio-ismb/ipynb/yeast.edgelist

[15] Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, Ziyue Huang, Qipeng Guo, Hao Zhang, Haibin Lin, Junbo Zhao, Jinyang Li, Alexander J Smola, and Zheng Zhang. 2019. Deep Graph Library: Towards Efficient and Scalable Deep Learning on Graphs. *ICLR Workshop on Representation Learning on Graphs and Manifolds* (2019). https://arxiv.org/abs/1909.01315

[16] Boris Weisfeiler and Andrei A Lehman. 1968. A reduction of a graph to a canonical form and an algebra arising during this reduction. *Nauchno-Technicheskaya Informatsia* 2, 9 (1968), 12–16.

[17] wikipedia. [n.d.]. *wikipedia: Confusion matrix.* Retrieved July 19, 2020 from https://en.wikipedia.org/wiki/Confusion_matrixb

[18] Xiaomei Wu, Lei Zhu, Jie Guo, Da-Yong Zhang, and Kui Lin 2006. *Prediction of yeast protein–protein interaction network: insights from the Gene Ontology and annotations.* Retrieved June 5, 2020 from https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1449908/

[19] Mu Zhu. 2004. Recall, precision and average precision. (09 2004).