המכללה האקדמית להנדסה בכרמיאל

בראודה

Software Engineering Department
Braude College

Capstone Project Phase B

GAN Graph Embeddings
23-1-R-5

Advisors:
Dr. Renata Avros and Prof. Zeev Volkovich

Authors:

Tzachi Ben Yair email: tzachibenyair@gmail.com

Eden Amir: eden1amir@gmail.com

Git repository:

https://github.com/edenamir/DynGan.git

Table of Contents

1. Project Review and Process Description

1.1 introduction

Phase A of our project focused on the application of the DynGAN model for link and graph prediction in the context of citation evaluation.
The DynGAN model, which combines generative adversarial networks (GANs) and long short-term memory (LSTM) networks, is employed to embed dynamic networks and predict the likelihood of future connections between scientific papers through citations. Phase A forms the foundation of our project, setting the stage for further analysis and exploration in phase B.

In this phase, we utilized a sequence of GANs to capture the temporal transitions in the dynamic network, while LSTM networks are employed to capture the evolution of the graph over time. By training the model on the iCite Database Snapshots, which provides bibliometrics and publications metadata, we aim to teach the methodology to differentiate between genuine and fake citations and references in articles.

We implemented the above methodologies in the developed software. The program provides the ability to predict future connections between nodes in a dynamic network. The future connections represent future references between two articles. This program allows the user to choose a database and a time interval, (the time passed between two snapshots of the network), receive the future graph and some statistical data about the prediction correctness.

1.2 The Algorithm

The DynGAN [1] employs the Graph-GAN framework, which combines generative and discriminative models to enhance performance in tasks like link prediction and node classification.
Generator G, generates "fake" samples to deceive Discriminator D, which aims to distinguish real and fake samples. Graph-GAN [2] introduces a novel function called graph SoftMax for the generative model. The DynGAN and DynGAN-LSTM models build upon Graph-GAN as the foundation for their implementation.

The primary objective is to capture temporal changes in the structural properties of the graph G_t . t is the graph snapshot at time t. These changes are instrumental in creating a stable, low-dimensional representation of the graph vertices. After training the framework, the parameter θ_g represents the graph's embedding.

The generator, G, employs a SoftMax activation function for all vertices in V. It aims to estimate the true connectivity distribution and selects vertices from V that have a high probability of being neighbors of V_c, the current selected vertex.

The process of vertex selection involves constructing a breadth-first search (BFS) tree from V. V_c is selected as the current vertex, and random samplings of adjacent vertices are accumulated. The random walk process concludes when a vertex V_i visits its parent, and V_i is then chosen as the generated vertex.

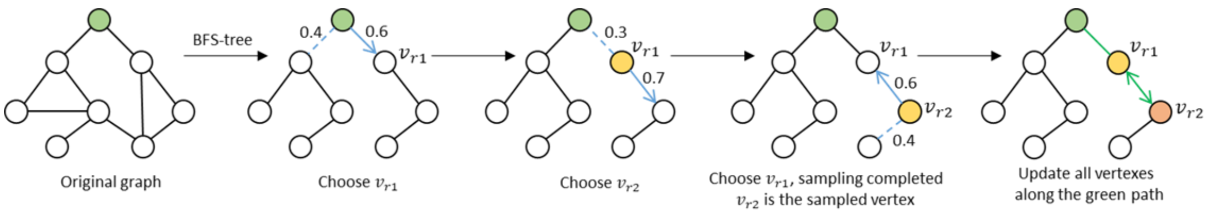


Fig 1: visual representation of the sampling process

Once samples of immediate neighbors from the true distribution and samples of potentially non-immediate neighbors from the generator are obtained, the discriminator's objective is to maximize the correct assignment of labels to all the samples.

On the other hand, the generator aims to minimize the discriminator's accurate labeling of the samples. In each iteration, it adjusts θ_g to shift its estimation of the true distribution. The change in parameters increases the discriminator's likelihood of labeling the generated samples as immediate neighbors. In other words, vertices with a higher probability of being labeled as non-immediate neighbors become less likely to be sampled by the generator through gradient descent on θ_g.

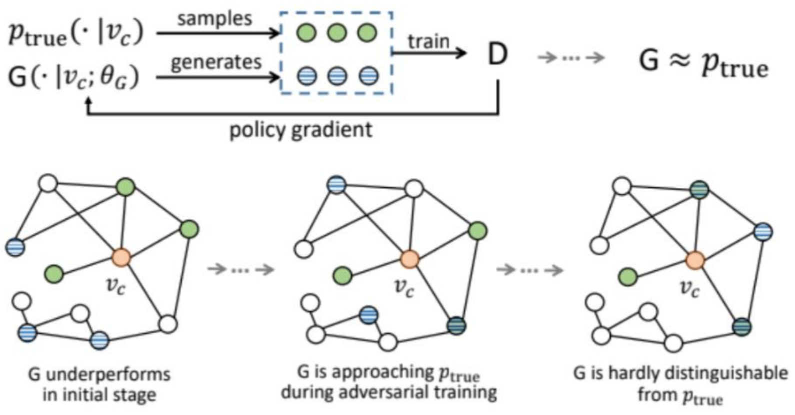

$p_{\text{true}}(\cdot|v_c)$
$G(\cdot|v_c;\theta_G)$
samples
generates
train
D
$G \approx p_{\text{true}}$
policy gradient
v_c
G underperforms in initial stage
G is approaching p_{true} during adversarial training
G is hardly distinguishable from p_{true}

Fig 2: Demonstration of the Graph-GAN [2].
The generator randomly selects a node v_c and associates it with a vector. The vector entries correspond to the true connectivity of the sampled nodes. As the generator updates itself, the generated and the true distributions become more and more alike making it harder for the discriminator to distinguish between them.

The DynGAN architecture consists of multiple Graph-GAN components connected in a sequential manner. Each GAN component receives two parameters: U_i, which represents the graph embedding for step i (generated from the previous step), and a graph snapshot for timestamp i+1. The initial embedding, U_{raw}, is a random embedding. This architecture enables the model to handle evolving graphs by retaining the weights from previous timestamps. Furthermore, the model produces a stable embedding as U_i is initialized with the embedding result from the previous timestamp, U_{i-1}. These advantages facilitate faster convergence during the training process with a smaller number of iterations. Although DynGAN is well-suited for link prediction, it struggles in graph prediction tasks. Its limited capacity to learn from previous time step data hinders its ability to capture temporal sequences in the network.

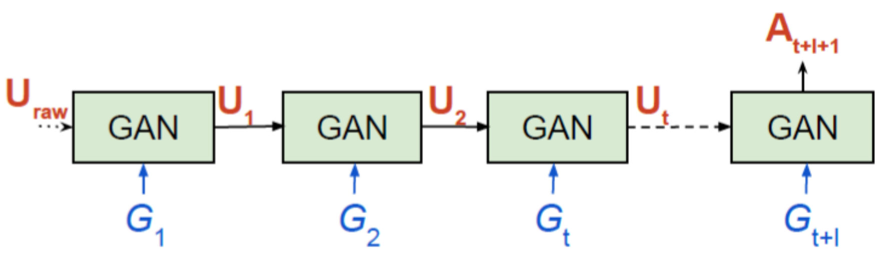


Fig 3: Architecture of DynGAN

In contrast, DynGAN-LSTM consists of multiple Graph-GAN components connected to a sequence of LSTM cells. Each sequence of LSTM cells is linked to the subsequent sequence, as depicted in Figure 4. This interconnection between cells enables the embedding, u_{v_i}, to capture both temporal and structural patterns for each vertex, v_i. Such information is crucial for predicting the vertex v_{i+1}. Each GAN component receives a graph snapshot, G_i, represented as a node adjacency matrix, A_i, for the time step i. The loss function penalizes incorrect reconstruction of edges at time t+l+1 by adjusting the parameters and leveraging the embeddings from previous time steps. This integration of LSTM cells enhances the model's ability to capture temporal dynamics and structural patterns, making it more suitable for graph prediction tasks.

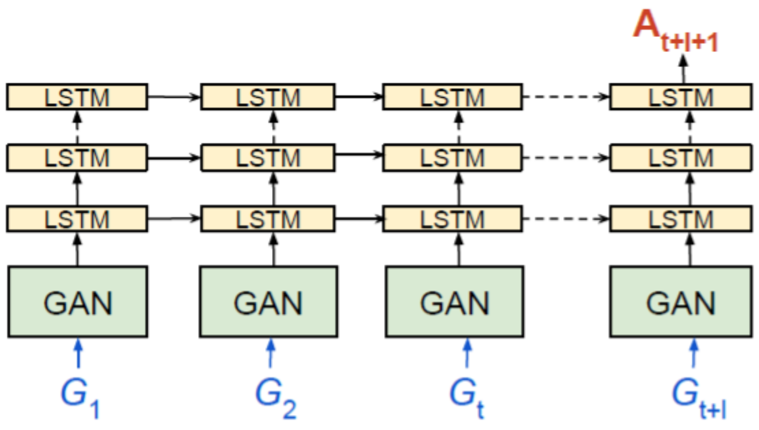


Fig 4: Architecture for DynGAN-LSTM

1.3 Research process

In addition to the research process conducted during phase A of the project [1] (refer to paragraph 4.1), we commenced the implementation phase by gaining a comprehensive understanding of the Graph-GAN code [4]. The code necessitated the utilization of Anaconda, an open-source distribution platform specifically designed for Python and tailored for scientific computing libraries and packages. To execute the code, we employed the Spyder IDE. Furthermore, we familiarized ourselves with various Python libraries such as numpy and networkx, which were crucial for generating the predicted graph.

The initial challenge entailed resolving existing bugs within the source code. Another obstacle involved adapting the code to serve as the foundational component for DynGAN. This adaptation involved establishing connections between the outputs of one Graph-GAN component and the inputs of the subsequent Graph-GAN in the sequence. However, the most significant challenge encountered was the extensive running time of the program. The training process for each individual GAN alone required nearly 12 hours to complete. Additionally, the downloading and loading of each snapshot, which contained approximately 40 million citations, consumed a substantial amount of time. To mitigate this issue, we utilized Google Colab for the latter task, aiming to reduce the overall execution time. To assess the project's performance in link prediction, we trained the model using only 85 percent of the edges in each snapshot. Subsequently, we evaluated whether the model accurately predicted the remaining 15 percent. The predicted results were then subjected to analysis using a confusion matrix.

1.4 Results

The outcomes of the link prediction are derived from the evaluation process. This process entails utilizing two files as data sources subsequent to training the model. The first file contains a collection of edges that have been removed from the snapshot, while the remaining edges are considered false. The objective of the evaluation is for the model to accurately identify and include the removed edges while disregarding the false ones. The edge selection process is conducted as follows:

Each node is represented as a vector within the embedded matrix. The connectivity probability between nodes in the edges file is influenced by the dot product of the corresponding nodes in the matrix. These probabilities are then stored to correspond with the respective edge positions in the edges list. Subsequently, the median of all probabilities is computed. Moving forward, the edges list is traversed, and for each edge, if the connectivity probability surpasses the median value, the model adds the edge to the network. Otherwise, the edge is not included.

To assess the results, a confusion matrix is employed, which is a well-known method for evaluating the performance of a classification algorithm. The key components of the confusion matrix include:

True Positive (TP): The number of predictions where the classifier correctly identifies the positive class as positive.

True Negative (TN): The number of predictions where the classifier correctly identifies the negative class as negative.

False Positive (FP): The number of predictions where the classifier incorrectly identifies the negative class as positive.

False Negative (FN): The number of predictions where the classifier incorrectly identifies the positive class as negative.

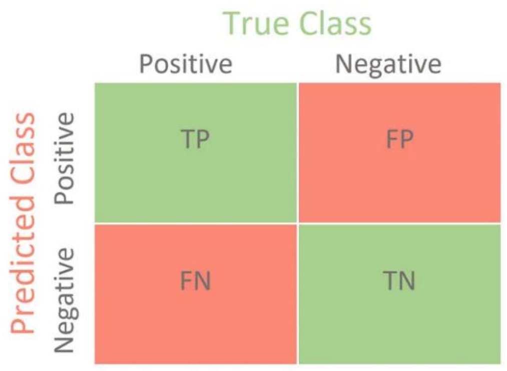


Fig 5: Confusion matrix

Out of 1101 real edges, 1101 fake edges this were the model results

TP	TN	FP	FN
577	587	522	523

Since we possess knowledge regarding the actual edges in the final snapshot, we can evaluate the model's results using the aforementioned methodology. Furthermore, we calculated the accuracy, precision, and recall of the model, defined as follows:

Accuracy: The proportion of correctly predicted classes (both positive and negative) out of the total classes.

$$(1)\frac{TP+TN}{TP+TN+FP+FN}=0.525$$

Precision: The proportion of correctly predicted positive classes out of the total classes predicted as positive.

$$(2)\frac{TP}{TP+FP}=0.525$$

Recall: The proportion of correctly predicted positive classes out of the total actual positive classes.

$$(3)\frac{TP}{TP+FN}=0.525$$

1.5 Conclusions and discussion

The obtained results are better than those presented in the referenced paper [1]. There is a discrepancy of 6% favoring our results in terms of the accuracy index. Our implementation achieved an accuracy of 0.525, whereas the paper reported an accuracy of 0.465. It can be inferred that this difference is attributable to the modifications made in our code. To enhance runtime efficiency, we made two primary adjustments:

1. Data Size: The original article utilized 733 network snapshots, encompassing 7,716 nodes and 26,467 edges in the final snapshot. In our implementation, we reduced the data to three snapshots, consisting of 6,218 nodes and 7,178 edges in the last snapshot.

2. Calculation Scale: The vector representation of each node in the original article comprised 50 features, whereas our implementation utilized 25 features. Additionally, the paper employed 50 components of Graph-GAN, whereas we employed three components.

The main reason for the enhancement of the results is due to the fact that the article tested the DynGAN-LSTM model on the task of graph prediction and not link prediction. We used the model for link prediction. This is a much simpler task

Within the code, each epoch of every GAN component compute and updates the vector embedding of each node. By reducing the vector size, we achieved expedited runtime. However, it is important to note that the results indicate a non-linear relationship between data size, the number of calculations, and the output. It is evident that to attain superior results, training the model on a larger network, representing nodes with a greater number of features, and incorporating more GAN components are necessary. The primary objective of the project was to implement DynGAN-LSTM and produce comparable outcomes to the original article in predicting future citations between articles. Upon comparing the results, we can confidently assert that we have achieved this goal.

The majority of challenges encountered during the project were related to comprehending the GraphGAN code and its adaptation for the DynGAN-LSTM model. To overcome these challenges, we sought guidance from our advisors and meticulously studied the documentation of all functions utilized in the code to deepen our understanding. Furthermore, the article lacked explicit instructions on implementing the presented architecture, prompting us to engage in a trial-and-error process to closely replicate the original design. Notably, several attempts were made to optimize the LSTM component. The prediction process is influenced by various layers and parameters. We determined that the input to the LSTM model should correspond to the number of snapshots in the GAN model. Each node is represented by three sequential snapshot vectors, which yielded improved prediction results. As for parameter selection, we carefully examined similar implementations in the code and opted for those that yielded superior prediction outcomes.

Throughout the entire project duration, the two of us collaborated closely and performed all tasks together. While this approach may have slowed down the progress, it ensured a clear understanding for both team members. Although task division could have expedited certain aspects, it also carried the risk of creating knowledge gaps in different project areas.

2. User Documentation

2.1 User’s guide operating instructions

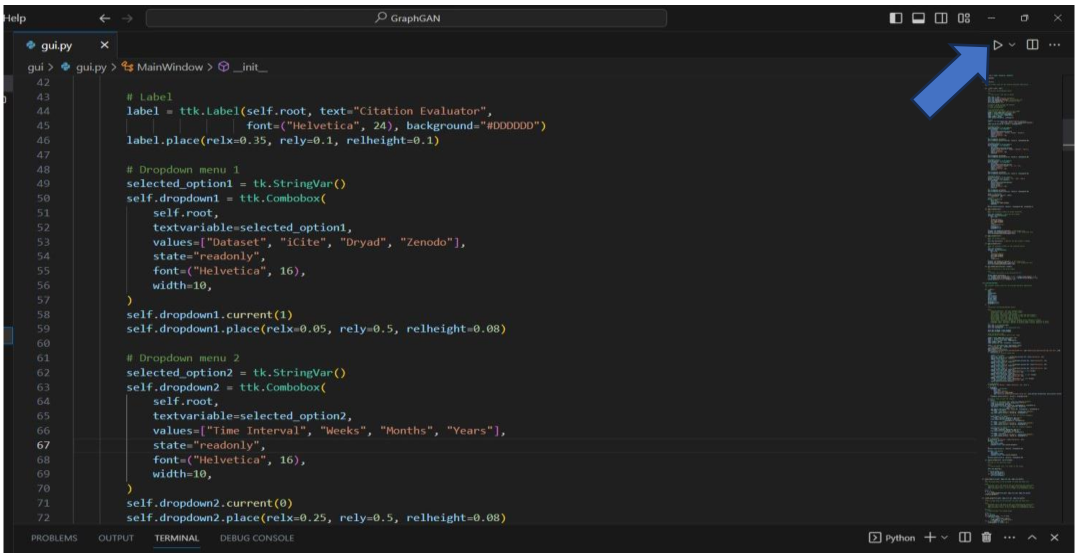
The system aims to forecast a new snapshot for a dynamic network. The current project focuses on distinguishing between real and fake citations in scientific papers from the iCite database. However, it is important to note that the system is general and can be applied to predicting the next snapshot in any dynamic network snapshots database. The implementation of the model is done in Python using the Spyder IDE, which is part of the Anaconda platform. For running the GUI use an IDE like Visual Studio Code.

Several essential Python libraries are utilized in this project, including numpy, networkx, scikit-learn, TensorFlow (compatible with version 1), pickle, and tqdm. For the user interface, tkinter and ttk are employed. Please make sure to install all of the packages for successfully running the code.

The software consists of two main components: model training and the GUI. The model training section is where the DynGAN-LSTM model is constructed and trained using the network snapshots. The iCite API is employed to obtain these snapshots, each represented as a list of node pairs: [node1, node2]. The GUI relies on the training results. In the first part, the predictions are written to a file, which then undergoes a post-processing phase for result extraction and visualization.

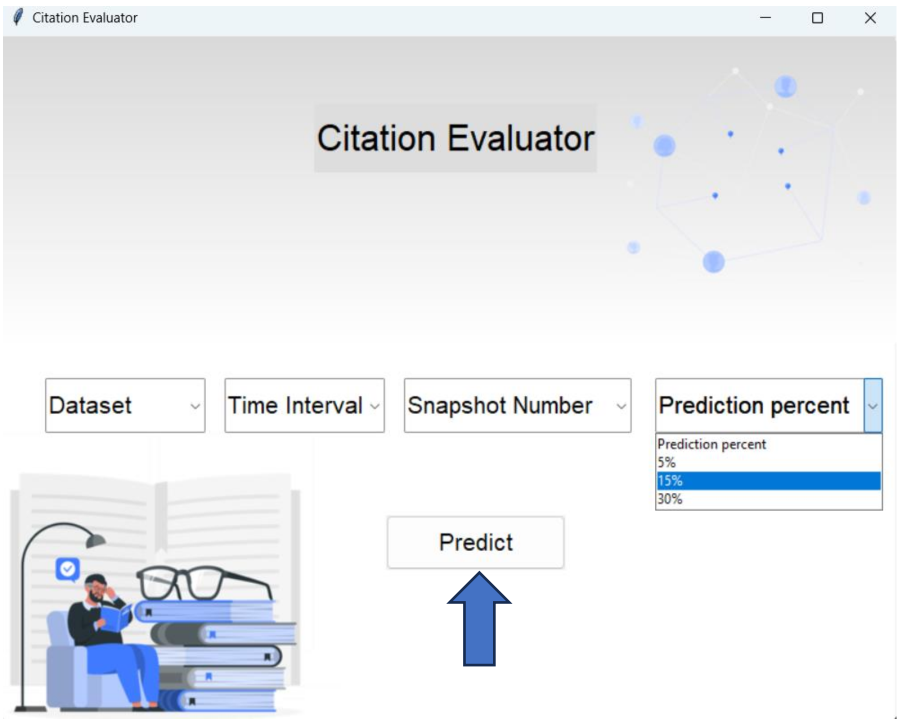
The software predicts the future snapshot and provides a metric for evaluating the accuracy of the prediction process.

2.2 Operating instructions



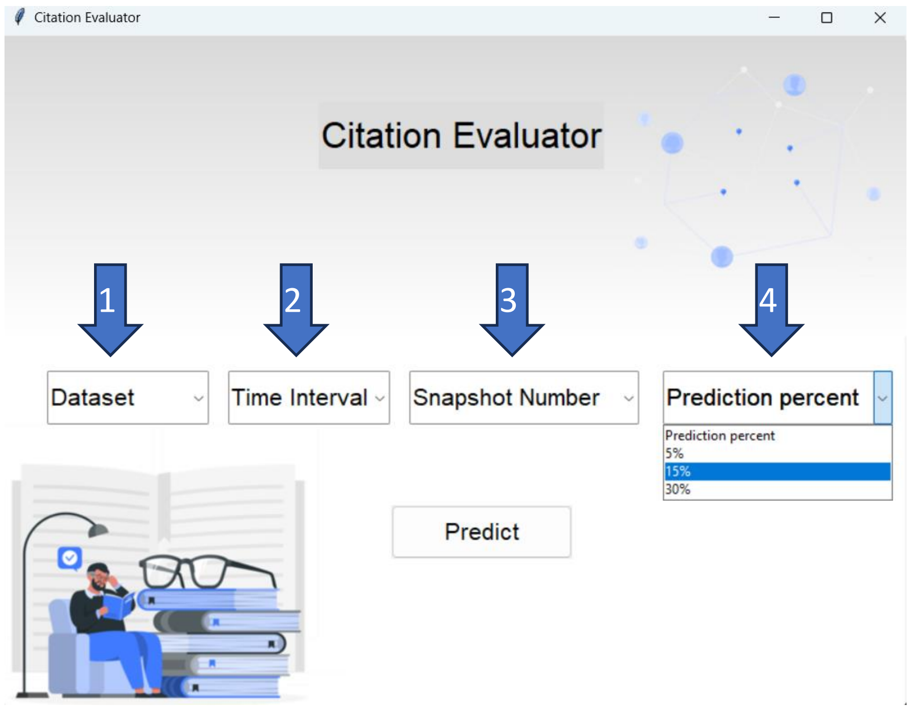
```python
        # Label
        label = ttk.Label(self.root, text="Citation Evaluator",
                          font=("Helvetica", 24), background="#DDDDDD")
        label.place(relx=0.35, rely=0.1, relheight=0.1)

        # Dropdown menu 1
        selected_option1 = tk.StringVar()
        self.dropdown1 = ttk.Combobox(
            self.root,
            textvariable=selected_option1,
            values=["Dataset", "iCite", "Dryad", "Zenodo"],
            state="readonly",
            font=("Helvetica", 16),
            width=10,
        )
        self.dropdown1.current(1)
        self.dropdown1.place(relx=0.05, rely=0.5, relheight=0.08)

        # Dropdown menu 2
        selected_option2 = tk.StringVar()
        self.dropdown2 = ttk.Combobox(
            self.root,
            textvariable=selected_option2,
            values=["Time Interval", "Weeks", "Months", "Years"],
            state="readonly",
            font=("Helvetica", 16),
            width=10,
        )
        self.dropdown2.current(0)
        self.dropdown2.place(relx=0.25, rely=0.5, relheight=0.08)
```

To run the project, please open an IDE that runs Python files. Here Visual Studio Code is shown. Load the project directories and click the run button.



The main window of the software. The user can define how the model will look like and change the parameters accordingly.

After all of the options are set. Click predict to go through to the next windows.

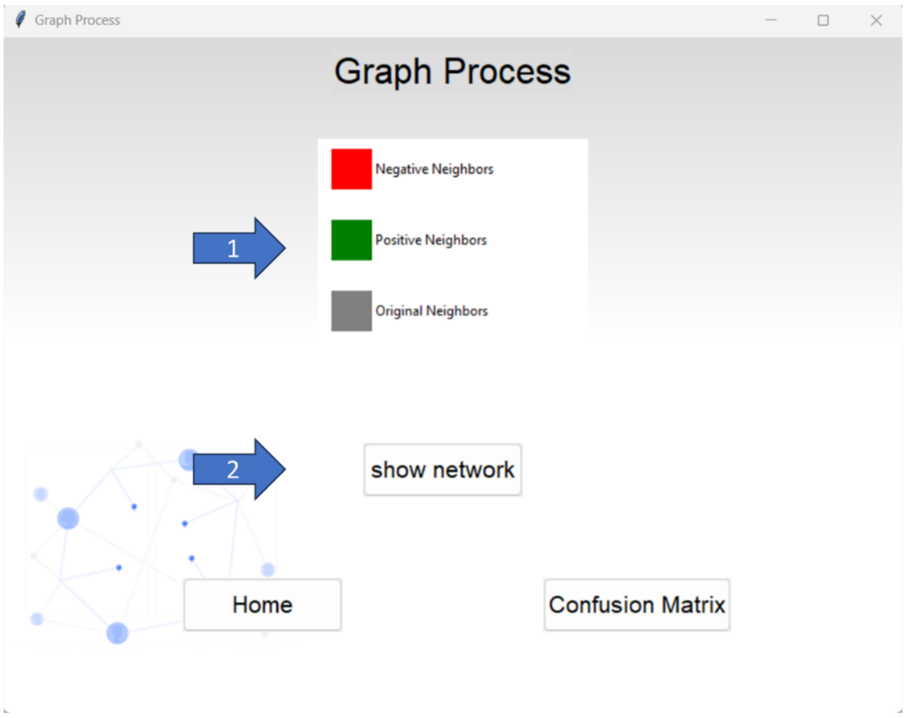


Arrow 1- option box "Dataset" for selecting the DB for the predicted network.

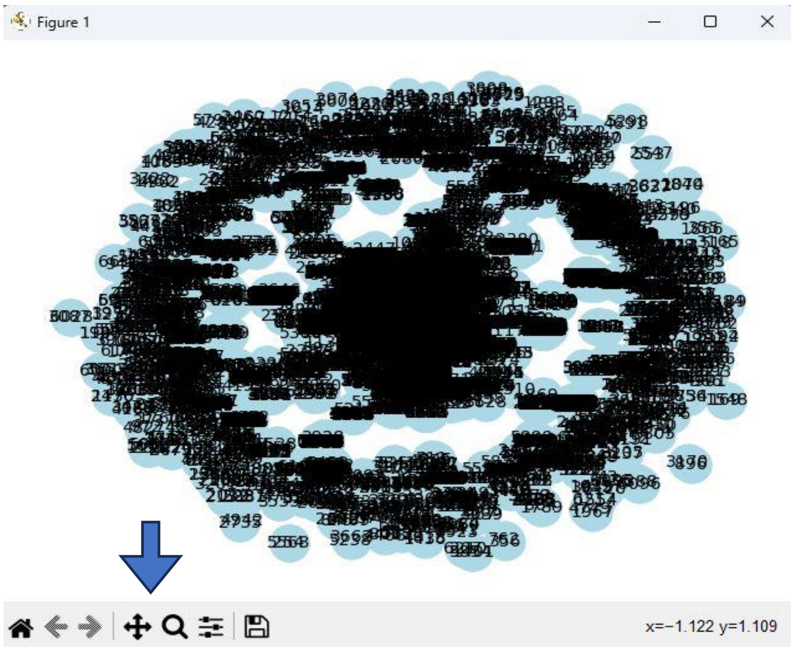
Arrow 2- option box "Time Interval" for selecting the time span between each snapshot.

Arrow 3- option box "Snapshot Number" for selecting the amount of GAN components i.e., the number of snapshots to consider in the prediction process.

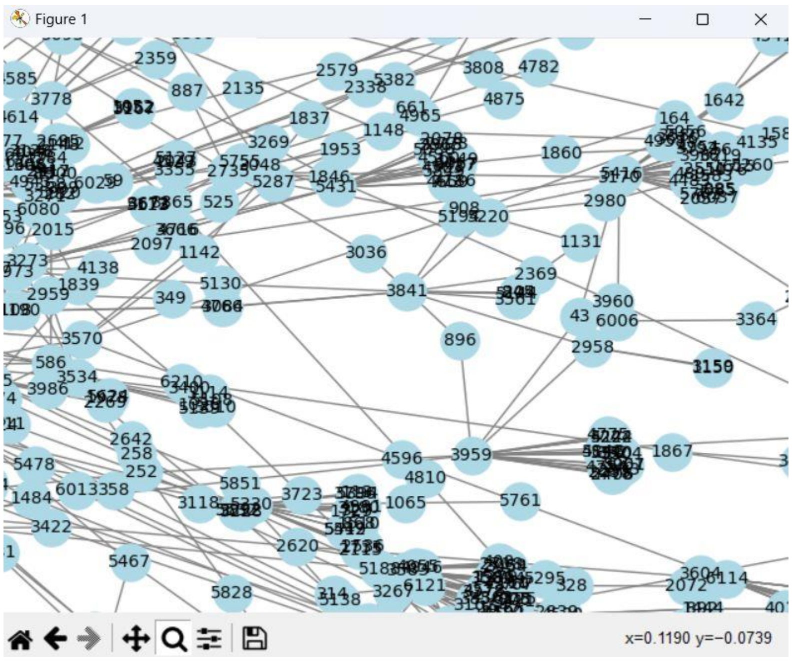
Arrow 4- option box "Prediction percent" for selecting the percentage of edges the user wants to predict.



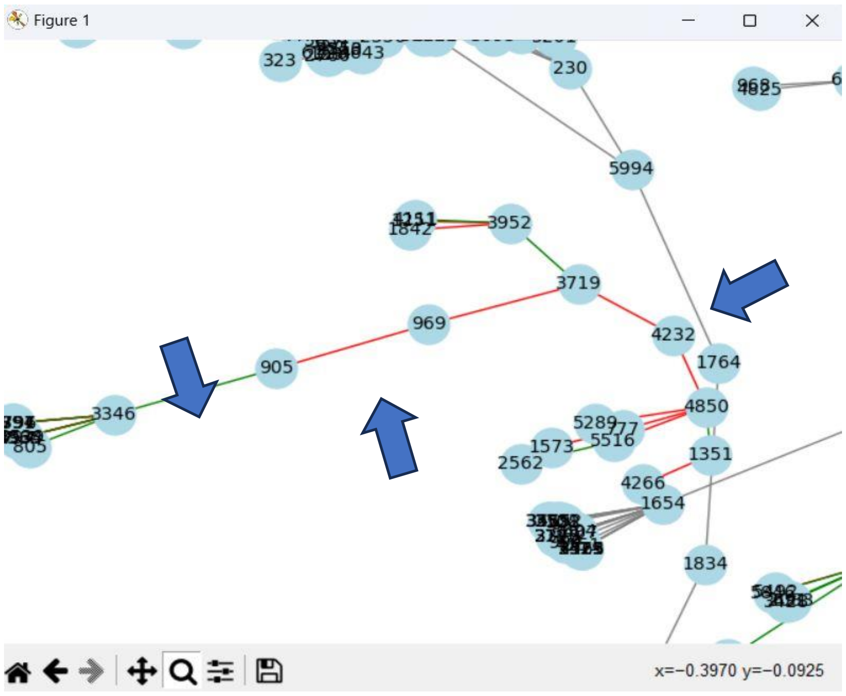
The second user interface, the top arrow shows a map indicating the colors of the different edges. The bottom arrow points to a button, showing the predicted network.



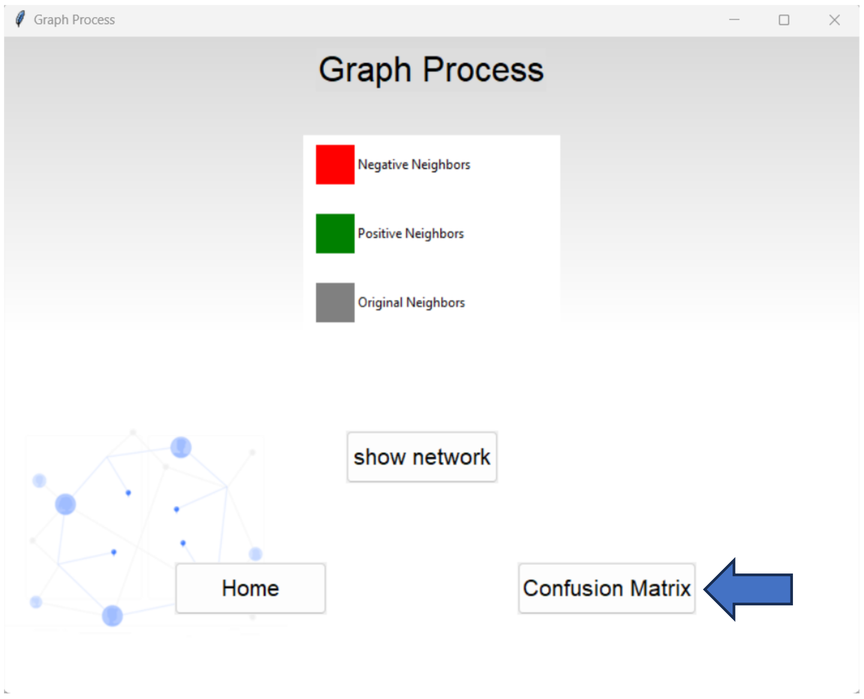
After pressing the show network button for the previous window, a new window appears with the visualization of the predicted network. Here, the arrow point on the navigation bar where the user can zoom in on parts of the graph, or choose to move the graph.



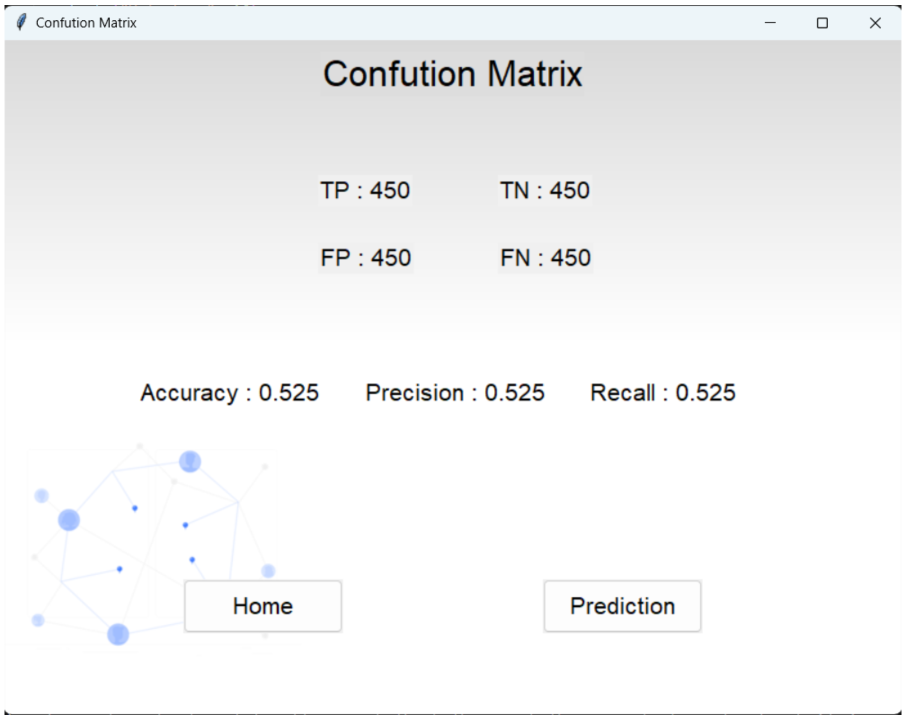
Example of zooming in on a random section of the network.



Another example of the zoom feature. Here the user can see the different colored edges. As the map suggest, red edges represent unrecovered edges that the model made the mistake to not incorporate in the future snapshot. To it, this are fake citations. The green edges represent recovered edges that the model correctly predicted. The gray edges are ones that appeared in the previous snapshot already.



The press the Confusion Matrix button to go through to the next window



Confution Matrix

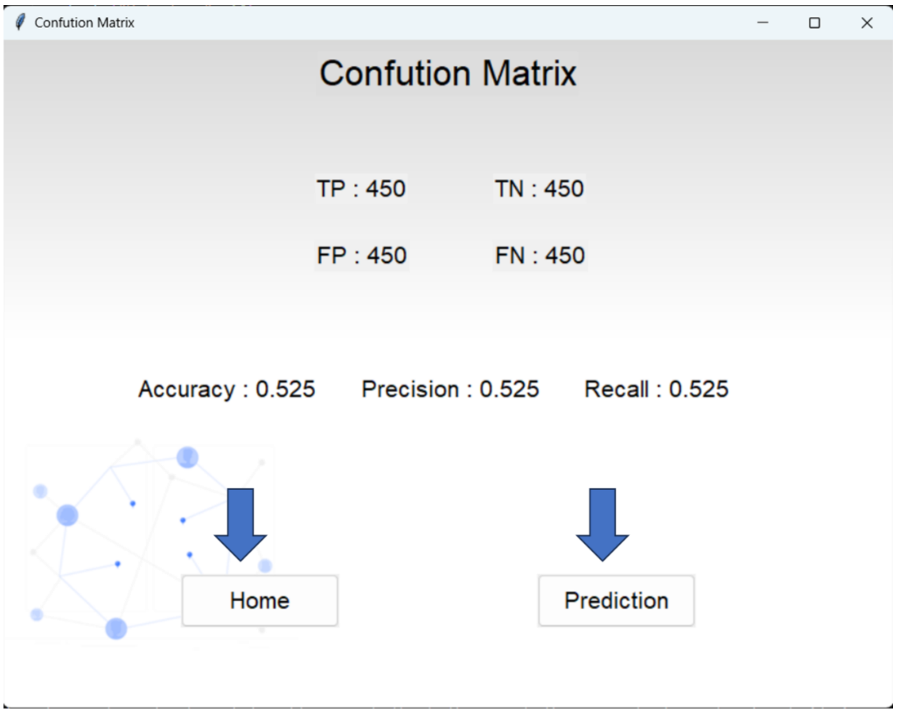
TP : 450   TN : 450

FP : 450   FN : 450

Accuracy : 0.525   Precision : 0.525   Recall : 0.525

This window presents the confusion matrix results. Is shows both the values of TP, TN, FP and FN, and the calculations for the accuracy, precision and recall values.

The calculations are based on the percentage of edges the user chose to predict.



Confution Matrix

TP : 450 TN : 450

FP : 450 FN : 450

Accuracy : 0.525 Precision : 0.525 Recall : 0.525

Navigation buttons to go back to the previous windows.

2.3 Maintenance Data

Directories name and information

Directory Name	Information
Data	Contains the initial text files required for the training process
Log	Contains checkpoints of the trained model in case the training cruses
Pre_train	Contains the initial embedding matrix. The matrix is initiated with random values.
Results	Contains all the prediction results and files to be used by the GUI
Src\GraphGAN	Contains the python files for the model
Src\evaluation	Contains the python files for the evaluation process of the model's prediction

To initiate the training process, the user is required to download the necessary snapshots. All functions for downloading and loading files are contained in the utils.py file. Following that, the user must provide an initial random embedding matrix. Finally, running the program is all that remains. The training process is time-consuming and dependent on both the number of GANs and the size of the network snapshot, as well as the computer's hardware

Once the model has completed training, the evaluation directory will contain a file named "testLPWithLSTM.txt." This file serves as the basis for all calculations in the post-processing phase.

With this file in place, the GUI component can be executed. It reads the "testLPWithLSTM.txt" file and compares its contents to the true labels of the edges (real or fake). Subsequently, it calculates the confusion matrix and other information to be presented in the GUI.

The following links direct you to the sources where you can download the Spyder IDE from:

Anaconda: https://www.anaconda.com/download

Spyder: https://anaconda.org/anaconda/spyder

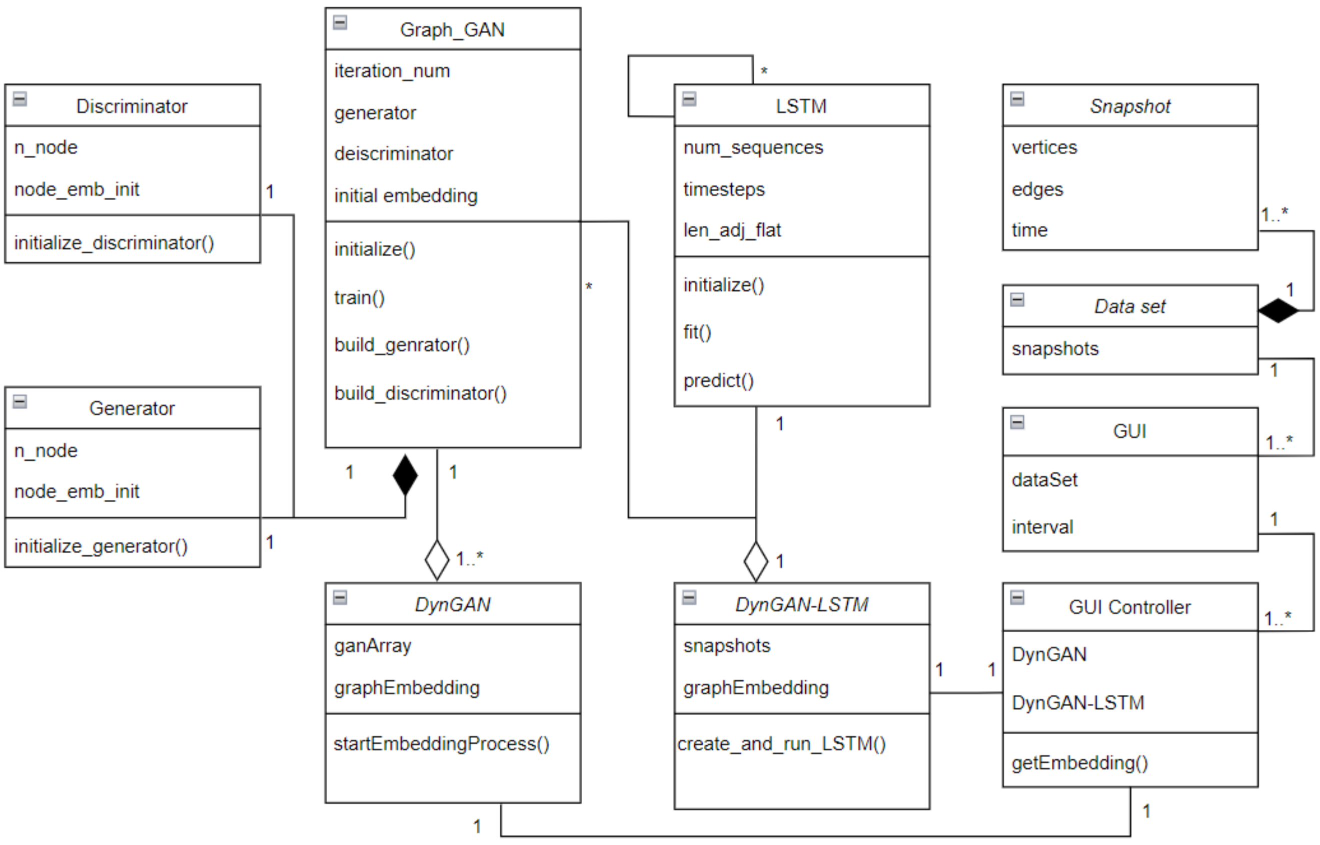


Fig 6: class diagram of the software

3. References

[1] Maheshwari, Ayush, et al. "DynGAN: generative adversarial networks for dynamic network embedding." Graph representation learning workshop at NeurIPS. 2019.

[2] Wang, Hongwei, Jia Wang, Jialin Wang, Miao Zhao, Weinan Zhang, Fuzheng Zhang, Xing Xie, and Minyi Guo. "Graphgan: graph representation learning with generative adversarial nets (2017)." *arXiv preprint arXiv:1711.08267*.

[3] Sarang Narkhede, https://towardsdatascience.com/understanding-confusion-matrix-a9ad42dcfd62

[4] Hongwei Wang, https://github.com/hwwang55/GraphGAN