Friend recommendation with Graph Neural Networks

Barát ajánlás Gráf Neurális Hálókkal

Group: Node0, Members: El-Ali Maya (BHI5LF), Simkó Máté (O3BMRX)

1. Introduction

Nowadays, networks are present in every aspect of life, from the very noticeable transport and social networks to the invisible web of the Internet, down to the microscopic networks of molecules making up the fabric of the world. Network analysis is a vast research field seeking to investigate these structures using statistical analysis and graph theory. Thanks to the information boom of the last decade, the size of the networks available for analysis has greatly increased, parallel to the rise of computational power, which enables researchers to process these networks on a scale never seen before.

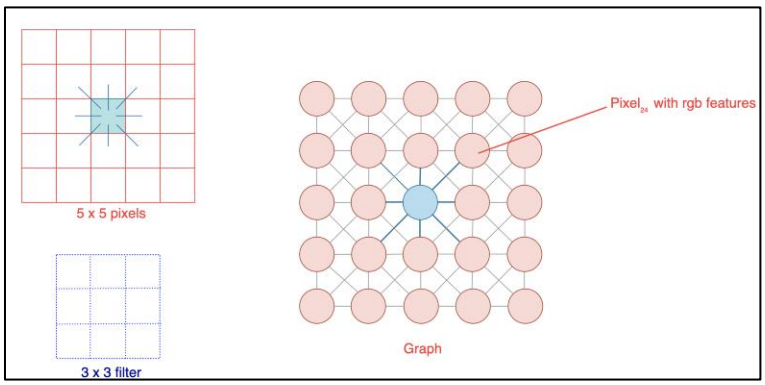
One of the vital applications of network analysis is friend recommendation, which seeks to enhance social networking platforms by intelligently connecting users with potential friends based on shared interests, activities, or mutual connections. These systems leverage complex algorithms and machine learning techniques to analyse user behaviour, interactions, and profile information, aiming to suggest connections that are not only relevant but also likely to foster meaningful relationships. With the advent of modern machine learning, several deep learning models have been proposed for operating on graph data for many different applications including friend recommendation.

Our goal was to develop a friend recommendation system by using Graph Neural Networks (GNNs) [1], that utilises data about the users, as well as, their connections. For this purpose, we used the Stanford Social Circles Dataset [2], specifically the Facebook dataset.

2. Background

2.1. Graph Convolutional Neural Networks

In today's world, many important datasets take the form of graphs or networks: knowledge graphs, social networks or recommender systems. Graph convolutional cetworks seek to harness the data present in the connections in these datasets. The structure of Graph Convolutional Networks most closely resembles Convolutional Neural Networks (CNN) [3] used mainly for analysing visual data. Convolutional neural networks can be understood as unique graph convolutional networks, since a picture is a special kind of graph where the vertices are the pixels, edges connect the neighbouring pixels, and the graph data stored in the pixels/vertices is the RGB values. This similarity can be seen



1. Figure Image data represented as a graph

in 1. Figure. Graph convolution works similarly to traditional convolution as information is gathered into the vertex from a given radius, much like filter size in image convolution.

2.2. Friend Recommendation

Graph convolutional networks have many different, the largest categories are the following:

- **Node/Vertex prediction**: This is used for assigning class labels or regression values to the nodes of the graph based on the labels or values of the neighbouring nodes. For example, finding fraudulent entities in a network in an e-commerce community falls into this category.
- **Edge prediction**: In this case, the network either predicts a value, for example, the weight, of the edges or classifies them. A special and common subsection of this is link prediction, when the network has to predict whether nodes have an edge between them. This use case is instrumental in the user recommendation systems used in the industry, we also use this technique for this project.
- **Graph prediction:** In this vast category, networks are used for classifying or assigning a regression value to whole graphs, for instance, calculating a chemical property of a molecule.

The friend recommendation task can be seen as an edge prediction task, where we have to predict new edges in the social graph.

3. Implementation

3.1. Dataset

The Stanford Social Circles dataset contains subsets of social media connection graphs (the nodes are the users, the edges are their connections) from different platforms, we used the Facebook dataset. The dataset also contains obfuscated a data about the users in the form of anonymised feature vectors. This means that we don't know what class the users belong to, but we can identify users in the same class. The dataset contains a large graph with more than 4000 nodes and 10 ego graphs. An ego graph is a subset of the original graph, which has a special node: the ego user, who is connected to all other nodes in the graph. Some attributes (number of nodes, edges, average node degree) of the ego graphs can be seen in 1. Table. The dataset contains different graphs, the smallest only consists of 59 nodes, while the largest contains more than a 1000. People tend to have a lot of Facebook connections, which is reflected in this dataset – the graphs have a lot of edges, the average node degree ranges from around 10 to 160. We chose to train separate networks on each ego graph.

Ego User	Number of nodes	Number of edges	Average node degree
0	347	5038	29.037464
107	1045	53498	102.388517
3980	59	292	9.898305
3437	547	9626	35.195612
686	170	3312	38.964706
1684	792	28048	70.828283
1912	755	60050	159.072848
698	66	540	16.363636
348	227	6384	56.246696
414	159	3386	42.591195

1. Table Statistics about the ego graphs

3.2. Tools and Libraries

The code was implemented in Python in Jupyter Notebooks, for the computationally heavy tasks we used Google Colaboritory for its GPU capabilities. For dealing with graph data, and graph convolutional networks we used two open-source python libraries:

- **NetworkX:** NetworkX is a free and open-source Python package for creating, manipulating and analysing complex networks. It offers a wide variety of data structures for different kinds of graphs, as well as easy conversion between different formats. We used this library for graph manipulation and visualisation
- **Deep Graph Library:** [4] Deep Graph Library(DGL) is a framework agnostic Python package for deep learning on graphs, which supports Pytorch, Tensorflow and Apache MXNet. It provides a user-friendly, yet flexible and powerful toolkit for building and training deep graph models. It offers state of the art models and modules as out-of-the-box solutions, as well as the bare building blocks for creating a GNN from scratch. The library supports training on large graphs, using multiple GPUs or machines making it suitable for performance intensive tasks. It also offers a comprehensive and easy-to-use documentation in addition to plenty of in-depth tutorials from the fundamentals to implementations of complex research papers. We used the tools offered by the library for creating GNN architectures, and for graph manipulation.

3.3. Data Preparation

Each ego graph is stored as an edge list in the dataset, we create DGL graph objects from these files. DGL graphs are capable of storing node and edge data, we store the absolute node labels, features and the ego user's id as node data. Usually for a friend recommendation task the dataset is split in time: the training dataset are the edges in the graph at a given time, test dataset is the edges at some later time. However, this was not available for us, so we split the graph's edges manually into train, validation and test sets in 7:1:2 ratio, by creating 3 graphs that only contain the relevant edges. We also create an inverse of each graph: *G(V,E)* is the original graph, *invG(V, invE)* is the inverse graph. Both graphs have the same nodes and in *invG* there is an edge between two nodes precisely when there isn't one in *G*. This was used to evaluate whether the model correctly predicts the missing edges as well. This methodology is based on an official DGL tutorial about link prediction. The feature matrix was already one-hot-encoded, so it didn't need further processing.

3.4. Model

Our solution for the edge prediction problem consists of the following steps: firstly, we compute the node embeddings using a GCN, and then we predict whether an edge exists by taking the dot product of its nodes and applying that score to the edges, finally the edges that have scores greater than a threshold get recommended as new connections.

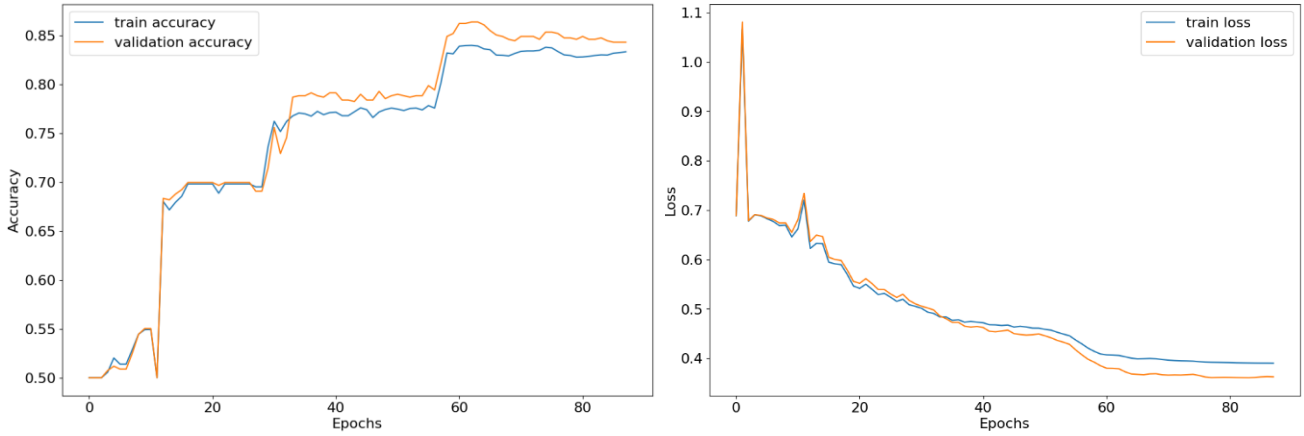
We created a customisable GCN network as a PyTorch module using the built in DGL GraphConv layer. The model contains sequential graph convolutional layers, with optional drop out layers after each. The number of layers and the size of them can be set in the model's constructor. The activation function of the graph convolution and the probability of the dropout can also be a customized, their default values are the ReLu function and 0 (no drop out) respectively. There is also an option for creating a narrowing network by setting the bottleneck parameter to true. In this case a layer will have half as many neurons as the previous one. This efficiently reduces dimensionality and captures essential features. We also define a DotPredictor also as a Pytorch module, that computes the predicted edge scores from the node embeddings.

3.5. Hyperparameter Optimisation

To attain better results, we optimised the parameters of the model and training function through experimentation. We performed a grid search where each combination of hyper parameters ran for 30 epochs, the combination with the least validation loss was chosen for the actual training. The optimised parameters are the following: hidden layer size in the networks (16,32,64,128), number of layers in the model (1,2,3,4,5), the probability of the drop out (0,0.3,0.5,0.8) whether the model should have a bottleneck architecture and the learning rate (0.01, 0.1, 0.001, 0.0001). The results of the hyperparameter optimisation are in the *hyperopt* folder with the *{ego_user}.csv* naming convention.

3.6. Training

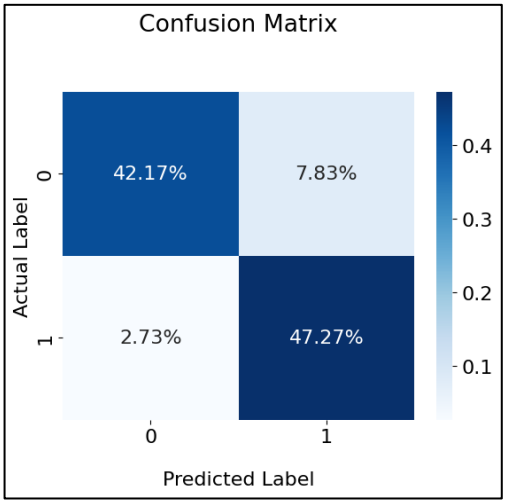
For training the models we use the Binary Cross Entropy loss and we define a custom function for calculating it using both the original and the inverse graph. Similarly, we defined a custom accuracy function. We use the Adam optimiser for training the model. In each epoch the loss and accuracy on the training and the validation graphs is calculated and saved. The model with the best validation loss is saved. The training includes an early stopping mechanism, which also aims to stop the model from overfitting, by stopping the training when for a certain number of epochs (the patience of the early stopping) there was no improvement in the validation loss. At the end of the training, we visualise the training curves: the change of the loss and accuracy on the training and validation graphs during the training. The training curves for the 414 ego graph's training can be seen in 2. Figure. The other diagrams can be seen in the notebooks.



2. Figure Training curves of the model trained on the 414 ego graph (accuracy: left, loss: right)

3.7. Evaluation methods and results

We evaluate the model on the test graphs that contain previously unseen edges, by calculating the node embeddings with the model, then taking the dot product of the embeddings, this gives us edge scores. We take the sigmoid of the scores and then find the best threshold by calculating the precision-recall curve for different thresholds and choosing the one with the best f1-score. After this we predict the edges using the threshold and calculate classification metrics (precision, recall, f1-score) and the confusion matrix. The confusion matrix of the model trained on the 414 ego graph can be seen on 3. Figure, the other matrices and metrics can be viewed in the notebooks.



3. Figure Confusion matrix of the model trained on the 414 ego graph

The models can be used for friend recommendation by using the calculated node embeddings. For a given user the dot product of its embedding is calculated with all other users' embeddings, the best ones are chosen for friend recommendation. These new relationships are also visualised, which can be seen in the *Visualisation* notebook.

4. Results

The final performance of the models can be seen in 2. Table.

Ego graph	Accuracy
0	86%
107	86%
1684	81%
1912	72%
3437	70%
3980	78%
414	89%
686	81%
698	93%

2. Table Accuracy of the models

5. Conclusion

With this project our goal was to create models that are capable of suggesting new connections to users based on their features and already existing connections. Our solution uses graph convolutional networks, because these networks can take advantage of the graph structure of the dataset. To achieve the best results, we created a customisable architecture and we optimised its parameters. We achieved the best accuracy on smaller graphs, however models trained larger graphs have also performed well. We implemented a function for the friend recommendation feature and created a visualisation method for it.

6. References

[1] T. N. a. W. M. Kipf, „Semi-supervised classification with graph convolutional networks,” *arXiv preprint arXiv:1609.02907,* 2016.

[2] J. a. L. J. Mcauley, „Discovering social circles in ego networks,” *ACM Transactions on Knowledge Discovery from Data (TKDD),* 2014.

[3] Y. a. B. L. a. B. Y. a. H. P. LeCun, „Gradient-based learning applied to document recognition,” *IEEE,* %1. kötet86, %1. szám11, pp. 2278--2324, 1998.

[4] M. a. Z. D. a. Y. Z. a. G. Q. a. L. M. a. S. X. a. Z. J. a. M. C. a. Y. L. a. G. Y. a. o. Wang, „Deep graph library: A graph-centric, highly-performant package for graph neural networks,” 2019.

[5] „Social circles: Facebook,” [Online]. Available: https://snap.stanford.edu/data/ego-Facebook.html.