Social Network analysis

COMP 4332 Project 2

Yuen Man Him, Tsang Hing
KiGroup 31

Project Overview:

The project focuses on the use of graph-based network mining techniques, specifically DeepWalk and node2vec, to understand and predict connections within a social network.

Training Pipeline:

Data Loader:

The datasets are initially stored in CSV format and loaded into memory using pandas. The edges are then used to construct a directed graph using NetWorkX's DiGraph class, which allows for the storage of edge attributes such as weights, crucial for the embedding techniques used later.

Once loaded, the data is transformed into a graph format:

- **Nodes** represent entities (users)
- **Edges** represent interactions between entities. These can be directed and weighted based on the frequency or strength of the interactions.

Random Walk Generator:

The core of our approach lies in generating random walks on the graph to capture the local and global structure. Two primary methods used are:

- **First-order Random Walks**: These walks consider only the immediate neighbors of a node. They are simpler and faster but miss out on capturing the broader context within the graph.
- **Second-order Random Walks**: These walks consider the neighbors of the neighbors, incorporating a broader scope of the node's context in the graph. This method is particularly useful for capturing community structures and is implemented using alias sampling to efficiently handle non-uniform sampling probabilities.

Network Embedding Models:

The random walks generated are then used to train node embedding models. Two popular methods are used:

- **DeepWalk**: This model uses purely first-order information. Random walks are generated and fed into a Skip-Gram model to produce node embeddings.
- **node2vec**: An extension of DeepWalk, allowing control over random walks' search space through parameters p (return parameter) and q (in-out parameter). This flexibility helps in better capturing the network's community structure.

Training:

Both models are trained using random walks generated from the network. Word2Vec (Skip-Gram model) is used to learn node embeddings that summarize the structural roles of nodes based on their local graph neighborhoods.

Validation and Testing:

Node embeddings are used to compute similarities (cosine similarity) between pairs of nodes. True edges and an equal number of generated false edges are used to evaluate the model using AUC-ROC metric, which measures the model's ability to distinguish between true and false connections.

Parameter Tuning:

By this point, based on what we know from these two models, we have decided that our final model should be node2vec. We use DeepWalk to determine the optimal values for "node_dim", "num_walks", and "walk_length". With the same "node_dim" set to 20, we discovered that ("num_walks" = 9, "walk_length" = 6) slightly outperforms the other values.

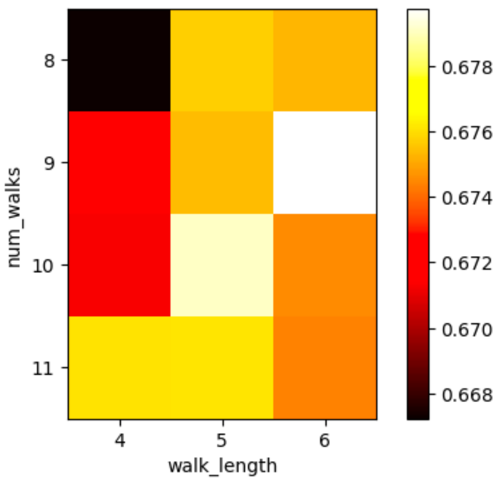


Figure 1: Heatmap of AUC scores when "node_dim" = 20

We believe the "walk_length" should hover around 6 as it correlates with the concept of six degrees of separation, which states that all people are six or fewer social connections away from each other. Similarly, setting "num_walks" to about 9 seems to store enough information—neither too little nor too much—for the model to yield a comparably accurate result.

Using this knowledge, we tested the effect of increasing the "node_dim" and found that doing so significantly enhances the AUC score.

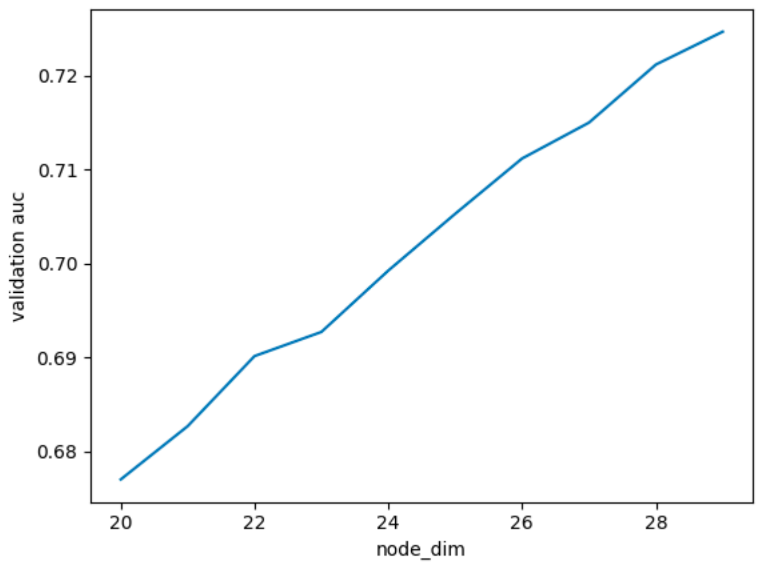


Figure 2: Line chart showing the effect of increasing “node_dim” against validation AUC

Subsequently, we employed node2vec, hoping that making the random walk biased could further increase the AUC score. At (“node_dim”, “num_walks”, “walk_length”) = (30, 9, 6), we found that (p, q) = (0.75, 0.85) outperformed the others. Therefore, we further reduced the size of the “p_list” and “q_list” to lower the computational demands for these hyperparameters and focused on tuning the “node_dim”. The “p_list” was reduced to [0.85, 0.90], and the “q_list” to [0.70, 0.75].

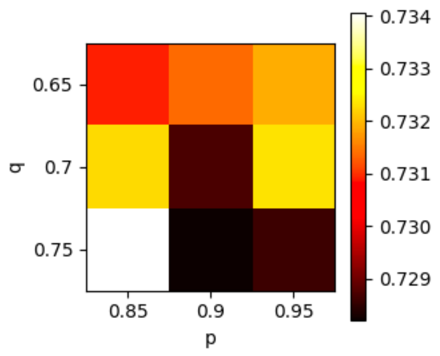
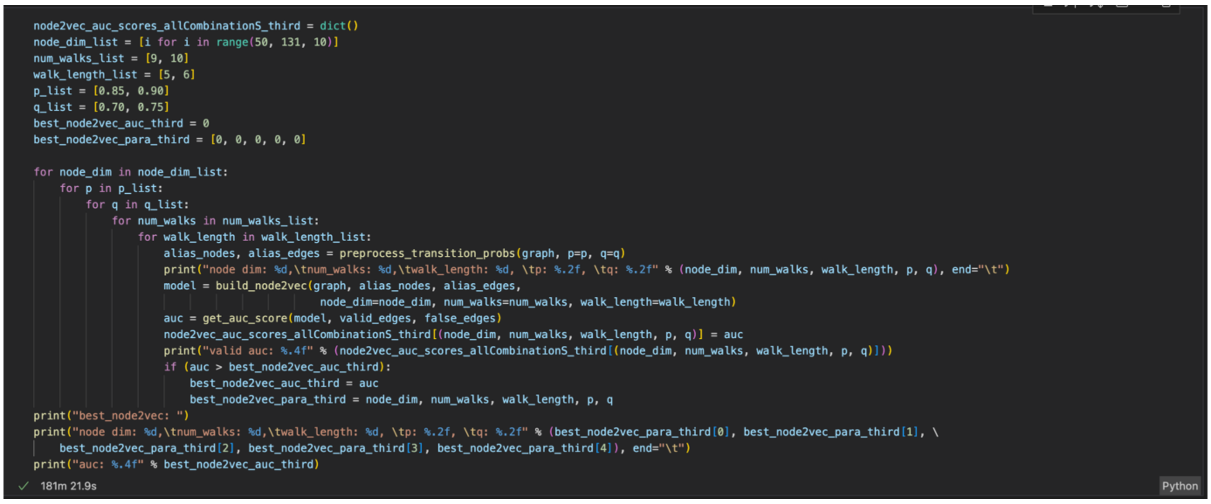


Figure 3: Heatmap of AUC score when (“node_dim”, “num_walks”, “walk_length”) = (30, 9, 6)

Keeping all other hyperparameters except “node_dim” around their optimal values that we previously identified, we think that allowing a bit of variation may explore more possibilities within limited resources and time, potentially yielding better results. We then conducted further training to find an optimal “node_dim” that achieves the highest score.



```python
node2vec_auc_scores_allCombinationS_third = dict()
node_dim_list = [i for i in range(50, 131, 10)]
num_walks_list = [9, 10]
walk_length_list = [5, 6]
p_list = [0.85, 0.90]
q_list = [0.70, 0.75]
best_node2vec_auc_third = 0
best_node2vec_para_third = [0, 0, 0, 0, 0]

for node_dim in node_dim_list:
    for p in p_list:
        for q in q_list:
            for num_walks in num_walks_list:
                for walk_length in walk_length_list:
                    alias_nodes, alias_edges = preprocess_transition_probs(graph, p=p, q=q)
                    print("node dim: %d,\tnum_walks: %d,\twalk_length: %d, \tp: %.2f, \tq: %.2f" % (node_dim, num_walks, walk_length, p, q), end="\t")
                    model = build_node2vec(graph, alias_nodes, alias_edges,
                                           node_dim=node_dim, num_walks=num_walks, walk_length=walk_length)
                    auc = get_auc_score(model, valid_edges, false_edges)
                    node2vec_auc_scores_allCombinationS_third[(node_dim, num_walks, walk_length, p, q)] = auc
                    print("valid auc: %.4f" % (node2vec_auc_scores_allCombinationS_third[(node_dim, num_walks, walk_length, p, q)]))
                    if (auc > best_node2vec_auc_third):
                        best_node2vec_auc_third = auc
                        best_node2vec_para_third = node_dim, num_walks, walk_length, p, q
print("best_node2vec: ")
print("node dim: %d,\tnum_walks: %d,\twalk_length: %d, \tp: %.2f, \tq: %.2f" % (best_node2vec_para_third[0], best_node2vec_para_third[1], \
    best_node2vec_para_third[2], best_node2vec_para_third[3], best_node2vec_para_third[4]), end="\t")
print("auc: %.4f" % best_node2vec_auc_third)
```

Figure 4: Code for finding the optimal “node_dim”

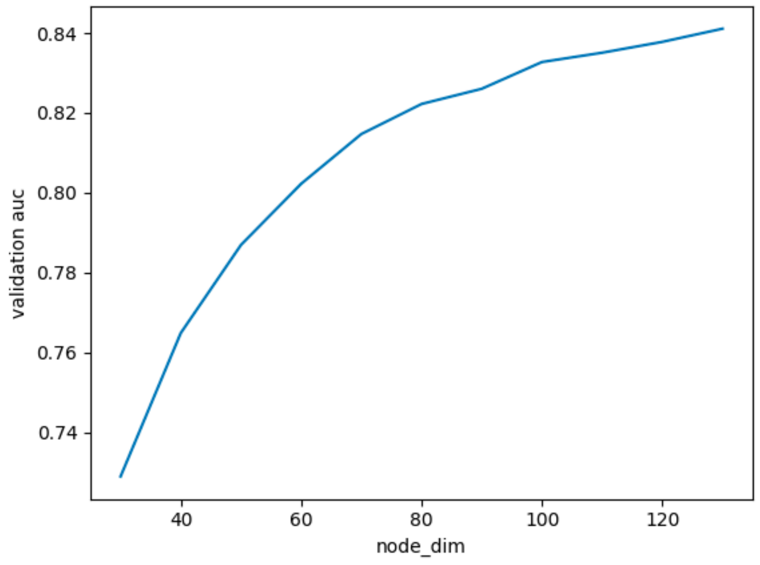
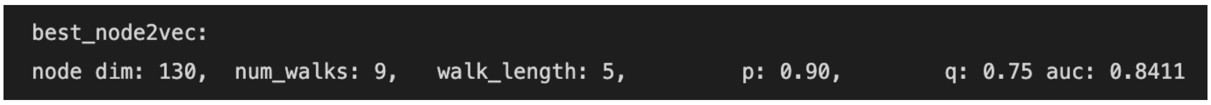


Figure 5: Line graph showing the increase of “node_dim” against validation AUC

A significant observation is the slower increase in AUC, a common phenomenon in deep learning models.

Meanwhile, we determined our hyperparameters with the highest validation AUC as follows:



```
best_node2vec:
node dim: 130,  num_walks: 9,   walk_length: 5,        p: 0.90,       q: 0.75 auc: 0.8411
```

Figure 6 Output of the cell from the figure 4 code

Further Enhancement:

We have already achieved a good AUC score with the node2vec model. Further enhancements were made by utilizing a subsampling method.

We iterate over the random walk, and for each position up to "length+1", we gather nodes spaced apart by "length" steps. Eventually, we return a list of these spaced-apart node groups, which we call transformed walks. Word2Vec takes these transformed walks as input.

The transformation effectively changes the sequence and grouping of nodes that the Word2Vec model will learn from, emphasizing relationships between nodes that are a fixed number of steps away from each other rather than consecutive steps. This can help capture longer-range interactions in the graph that might be missed in traditional contiguous sampling.

We chose "num_skips" = 5 for subsampling, and other parameters remain the same as shown in Figure 6. We obtained a validation AUC of 0.8550, which is slightly better than the standard node2vec model.

Evaluation:

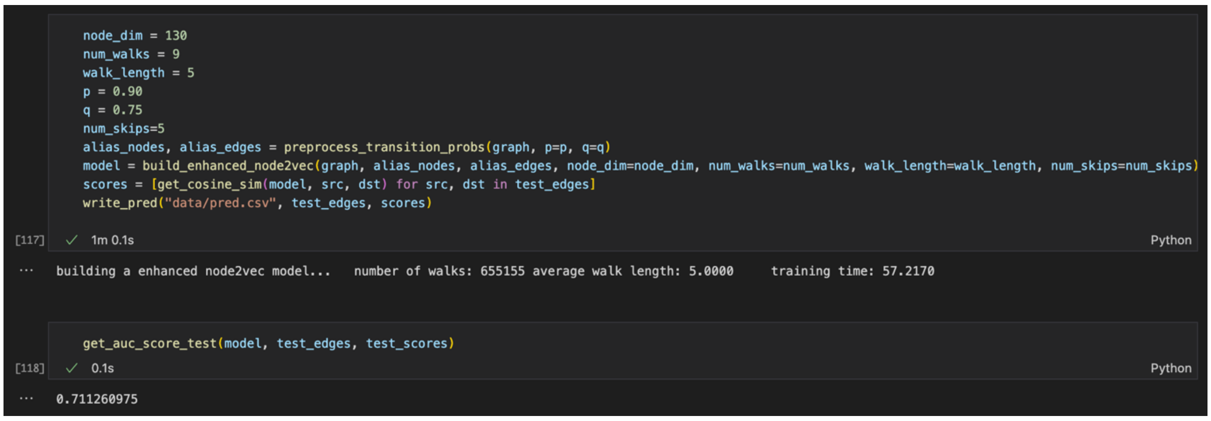
We utilize the following chosen hyperparameters for the testing the models:

Name	Values
node_dim	130
num_walks	9
walk_length	5
p	0.9
q	0.75
num_skips	5

Comparing results of different models:

Network embedding algorithm	Valid AUC	Test AUC
DeepWalk	0.8392	0.70170571375
node2vec	0.8411	0.70240713
Enhanced node2vec	0.8550	0.711260975

In conclusion, we have decided to choose the enhanced node2vec as our final model.



```
node_dim = 130
num_walks = 9
walk_length = 5
p = 0.90
q = 0.75
num_skips=5
alias_nodes, alias_edges = preprocess_transition_probs(graph, p=p, q=q)
model = build_enhanced_node2vec(graph, alias_nodes, alias_edges, node_dim=node_dim, num_walks=num_walks, walk_length=walk_length, num_skips=num_skips)
scores = [get_cosine_sim(model, src, dst) for src, dst in test_edges]
write_pred("data/pred.csv", test_edges, scores)
```

```
[117] ✓ 1m 0.1s                                                                    Python
··· building a enhanced node2vec model...   number of walks: 655155 average walk length: 5.0000     training time: 57.2170
```

```
get_auc_score_test(model, test_edges, test_scores)
```

```
[118] ✓ 0.1s                                                                       Python
··· 0.711260975
```

Figure 7 Final model code with output