Twitch Social Network Analysis

Jayesh Prasad Anandan (janandan@iu.edu)[1]*, Shivani Pal (shipal@iu.edu)[1]*, Himani Ravindra Borana (hborana@iu.edu)[1]*,Bhakti Patrawala (bhpatra@iu.edu)[1]*, Laxmikant Maheshji Kabra (laxkabra@iu.edu)[1]*

Abstract

Twitch has become one of the largest and the most popular video game streaming platforms. It has thus formed a unique social network among its users, ie, gamers. Our aim in this project is to to analyze and perform various tasks such as classification, prediction, clustering, and visualization on Twitch Gamers Social Network Dataset using Graph Machine Learning Models. We use various graph machine learning models such as Graph Convolutional Networks (GCN), Graph Attention Networks (GAT), and Graph Neural Networks (GNN) in order to do so. Our results show that these graph machine learning models can sometime effectively extract useful information from the Twitch Gamers Social Network Dataset and perform well on the given tasks, on the other hand may not perform very well in some cases.

Keywords

Graph Analysis, Twitch network, Centrality, Community Detection Classification, Link Prediction

[1] *Luddy School of Informatics, Computing, and Engineering, Indiana University, Bloomington, IN, USA*

Contents

1. Introduction

The aim is to analyze the structure and other characteristics of Twitch social network by understanding the dynamics of mutual follower relationships among them. Here are some methods we used to achieve our objective:

1. Prediction: Link prediction methods are used to spot potential follower relationship between the Twitch users, by predicting future connections between the nodes. Essentially, link prediction conveys whether a link exists between two nodes. The nodes in this case are the users of the Twitch dataset. Additionally, it is used to provide new recommendations to the users based on their viewing history and preferences by pattern in users. We have used the Graph Convolution Network (GCN) and Graph Sage algorithms to perform link prediction on the users of the Twitch streamers dataset.

2. Clustering: Using community detection algorithms, which follow hierarchical clustering to determine various communities within data to group users based on similar properties.

3. Classification: Using various machine learning algorithms, we performed various classification tasks such as affiliate status classification, explicit content streamer identification, and user lifetime estimation on the Twitch gamers social network.

The tasks we performed are:

1. Link Prediction
2. Dead account classification
3. Explicit content classification
4. Affiliation status classification
5. Language Classification
6. Lifetime Prediction
7. Community Detection

2. Related work

1. *Thomas N. Kipf, Max Welling* in their paper *(Semi-Supervised Classification with Graph Convolutional Networks)* [1] has furnished us with the use of Graph Convolutional Networks in Node Classification.

2. *William L. Hamilton, Rex Ying, Jure Leskovec* in their paper *(Inductive Representation Learning on Large Graphs)* [2] has introduced us to the Graph SAGE model.

3. *Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, Yoshua Bengio* in their paper *(Graph Attention Networks)* [3] has introduced us to the Graph Attention model.

4. *Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, Qiaozhu Mei* in their paper *(LINE: Large-scale Information Network Embedding)* [4] has helped us to understand the Graph Embedding and Negative sampling for Link Prediction.

5. *Alexey Dosovitskiy, Philipp Fischer, Jost Tobias Springenberg, Martin Riedmiller, Thomas Brox* in their paper *(Discriminative Unsupervised Feature Learning with Exemplar Convolutional Neural Networks)* [5] talks about the Negative Sampling and Transfer Learning concepts.

3. Method

To analyze the characteristics of the network, we visualize the following:

1. Degree Distribution: We plot a graph showing the degree distribution of the network. The algorithm is given below:
   Degree_Distribution(G):
   (a) Initialize an empty list called ‘degree‘.
   (b) For each node v in G: Calculate the degree of node v in G and append it to the ‘degree‘ list.
   (c) Plot the degree distribution using the ‘degree‘ list as input with x-axis as ”Degree (k)” and the y-axis as ”Number of nodes with degree k (Nk)”.

2. Clustering Coefficient Distribution: The algorithm for finding the local cluster coefficient of all vertices and plotting them is given as follows:
   Clustering_Analysis(G):
   (a) var clust: Calculate clustering coefficient for each vertex in G using nx.clustering(G)
   (b) local_clust_coefficient: List of clustering coefficients for each vertex
   (c) For each vertex v in G: Add clust[v] to local_clust_coefficient
   (d) avg_clust_coefficient: Sum of local_clust_coefficient divided by the number of vertices in G
   (e) Plot the distribution of the clustering coefficient setting the y-axis scale to the logarithmic scale.

3. Shortest Path Distributions: We find out the shortest path distribution for the 5 largest connected components. The algorithm is as follows:
   (a) Sort connected components of graph G in decreasing order of size and store in cc_sorted.
   (b) Set topcc to the minimum of length of cc_sorted but since the graph is large in our case, we only keep 5.
   (c) For i from 0 to topcc-1 do the following:
      i. Set cc to the i-th connected component in cc_sorted.
      ii. Create a subgraph cc_graph of G using the vertices in cc.
      iii. If the size of cc is greater than 30,000, then we find single-source shortest paths. For j from 0 to 9 do the following:
         A. Compute the shortest path lengths from the j-th vertex in cc to all other vertices in cc_graph using nx.single_source_shortest_path_length.
         B. Append the shortest path lengths to shortest_path_lens.
      iv. Plot the distribution of the shortest path length with the x-axis as the shortest path lengths (hops) and the y-axis as the number of paths.

4. Link Prediction: The link prediction task is carried out by using neural networks. Initially, we intended to use embedding methods like DeepWalk and Node2Vec in order to embed the edges dataset and then carry out link prediction. However, since the Twitch Streamers dataset is comparatively larger than most other general datasets such as the cora dataset, the embedding consumed an extensive amount of time without giving any significant results. Hence, we moved forward with the neural networks approach. We have used two algorithms namely the Graph Convolution Network(GCN) and the Graph Sage network to perform link prediction.
   (a) Graph Convolution Network - Graph convolution networks are popularly used to operate on Graphs which are a type of data structures that store entities and the relationships between them. Thus, GCN works on nodes and edges where nodes represent the features or attributes of the entities and the edges represent the relationships between these entities. The ability of GCN to learn a low-dimensional representation for each node in a graph that captures its local and global relationships with other nodes makes it a good candidate to perform link prediction.
   Following are the steps in GCN -
   (b) Graph Sage is a potent technique that efficiently captures the structure and characteristics of the graph while doing embedding. By compiling data from each node’s immediate vicinity, the embedding is carried out.

```python
class Net(torch.nn.Module):
    def __init__(self, in_channels, hidden_channels, out_channels):
        super().__init__()
        self.conv1 = GCNConv(in_channels, hidden_channels)
        self.conv2 = GCNConv(hidden_channels, out_channels)

    def encode(self, x, edge_index):
        x = self.conv1(x, edge_index).relu()
        x = self.conv2(x, edge_index)
        return x

    def decode(self, z, edge_label_index):
        return (z[edge_label_index[0]] * z[edge_label_index[1]]).sum(dim=-1)

    def decode_all(self, z):
        prob_adj = z @ z.t()
        #return the indices of a non-zero element
        return (prob_adj > 0).nonzero(as_tuple=False).t()
```

Figure 1. Neural Network Model with Encoder and Decoder

5. Following is the detailed explanation of the model -
   (a) The Net class is defined as a subclass of torch.nn.Module, which is the base class for all neural network modules in PyTorch. This means that Net inherits many useful methods and attributes from torch.nn.Module, such as the ability to define trainable parameters and compute gradients.
   (b) The __init__ method is a special method that initializes the Net object when it is first created. It takes three arguments: in_channels, hidden_channels, and out_channels, which specify the number of input, hidden, and output channels in the GCN layers.
   (c) Inside __init__, the two GCN layers are defined using the GCNConv class from the PyTorch Geometric library. The first layer takes in_channels as input and produces hidden_channels outputs, while the second layer takes hidden_channels as input and produces out_channels outputs.
   (d) The encode method takes as input a node feature matrix x and an edge index edge_index, and applies the GCN layers to compute a low-dimensional representation of the graph. The output is returned as a tensor x of size (num_nodes, out_channels).
   (e) The decode method takes as input the low-dimensional representation z computed by encode, and an edge label index edge_label_index, and computes a score for each pair of nodes in edge_label_index that indicates the likelihood of an edge between them. Specifically, it computes the dot product of the low-dimensional representations for the two nodes in each pair and returns a tensor of scores of size (num_edges,).
   (f) The decode_all method takes as input the low-dimensional representation z computed by encode, and computes a score for all possible pairs of nodes in the graph. It first computes a probability adjacency matrix prob_adj by taking the dot product of z with its transpose and then returns the indices of all non-zero elements in prob_adj as a tensor of size (2, num_edges).
   (g) Inside the nonzero method is called with the argument as_tuple=False to return a tensor of indices where the elements of prob_adj are non-zero. The resulting tensor is then transposed using the t method to produce a tensor of size (2, num_edges), where each column contains the indices of the two nodes that form an edge.
6. Negative Sampling - Negative sampling entails creating negative examples in which there is no connection between the nodes to contrast with the positive ones in which there is a connection. As a result, the model may be trained more effectively since it can learn to distinguish between links that are there and those that are not. In our code, we can see that the following line is performing negative sampling -

   ```
   neg_edge_index = negative_sampling(
   edge_index=train_data.edge_index,
   num_nodes=train_data.num_nodes,
   num_neg_samples=train_data
   .edge_label_index.size(1))
   ```

   This performs negative sampling to generate a set of negative edges for training. Negative sampling is a technique that generates fake edges (i.e., edges that do not exist in the original graph) in order to provide negative examples for the model to learn from.
   In particular, the negative_sampling function takes as input the edge index of the positive examples ($train\_data.edge\_index$), the total number of nodes in the graph ($train\_data.num\_nodes$), and the number of negative samples to generate ($train\_data.edge\_label\_index.size(1)$). It then randomly generates pairs of nodes that do not have a link between them (i.e., negative examples) and returns their edge index (neg_edge_index).

   The code then concatenates the positive examples and negative examples together to form the edge label index ($edge\_label\_index$) and edge label ($edge\_label$) for training the model. The loss function then measures the difference between the predicted edge labels and the actual edge labels and backpropagates the error through the model to update its parameters.
7. Node Classification: We are using Graph Convolution Network (GCN) for classifying the node features like dead account, language classification, affiliation status, and explicit content.
   (a) Dead Account Classification
      i. The Dead Account Node feature conveys whether the account is still active or dead with no content published or streamed in recent times as binary labels 0 and 1 where 0 represents that the account is still active and 1 represents the account is dead.

ii. Our input to the GCN will be only the Node Labels and the target column which here is the dead account column.
iii. Initially, we tried using the GCNConv Model with 2 layers of 16 units and default hyperparameters. The input is 1 node and the output is 2 classes with binary labels 0 and 1 meaning Live and Dead Account.
iv. Later, we used GraphSAGE as SAGEConv with a similar setting as the GCNConv. The input is 1 node and 2 layers with 16 units then the output is 2 nodes denoting two classes with binary labels 0 and 1.
v. Finally we compared the results of GCNConv and SAGEConv with the Graph Attention Network (GAT) with a similar model structure and the same parameters.

(b) Language Classification
i. The Language Node feature conveys the language in which the Twitch streamer is streaming the account. These are represented as string labels each representing the streaming language. The unique streaming languages are 21.
ii. Our input to the GCN will be only the Node Labels and the target column is the languages column.
iii. Initially, we tried using the GCNConv Model with 2 layers of 16 units and default hyperparameters. The input is 1 node and the output is 21 node with multi-class labels 0 to 20 denoting the different language classes.
iv. Later, we used GraphSAGE as SAGEConv with a similar setting as the GCNConv. The input is 1 node and 2 layers with 16 units then the output is 21 nodes denoting 21 classes.
v. Finally we compared the results of GCNConv and SAGEConv with the Graph Attention Network (GAT) with a similar model structure and the same parameters.

(c) Affiliation Status
i. The Twitch Social platform has an affiliation program wherein a User can join the program upon certain eligibility and get income out of the content he has posted by allowing advertisements on his posts. It's a binary label with 0 and 1 where 1 implies the user is affiliated.
ii. Our input to the GCN will be only the Node Labels and the target column is the Affiliation Status.
iii. Initially, we tried using the GCNConv Model with 2 layers of 16 units and default hyperparameters. The input is 1 node and the output is 2 classes with binary-class labels 0 and 1 denoting the affiliation status.
iv. Later, we used GraphSAGE as SAGEConv with a similar setting as the GCNConv. The input is 1 node and 2 layers with 16 units then the output is 2 nodes denoting 2 classes.
v. Finally we compared the results of GCNConv and SAGEConv with the Graph Attention Network (GAT) with a similar model structure and the same parameters.

(d) Explicit Content
i. The Explicit content label in the dataset speaks about the content streamed by a Twitch user. Some games can contain scenes with nudity and violence which can be termed as explicit to gamers who are younger in age. The Explicit Content is labeled as 0 and 1 where 1 implies the content streamed is explicit.
ii. Our input to the GCN will be only the Node Labels and the target column is the explicit content.
iii. Initially, we tried using the GCNConv Model with 2 layers of 16 units and default hyperparameters. The input is 1 node and the output is 2 classes with binary-class labels 0 and 1 denoting whether the content is explicit or not.

8. Lifetime Prediction: In order to predict the lifetime of Twitch gamers and their probability of continuing in the future, we implemented two graph neural network models, Graph Convolution Network (GCN) and SAGEConv. The account lifetime of each gamer was represented in days, showcasing the first and last day of their stream. To process this data, we converted the lifetime into years and divided it into 12 classes, which were fed into the two above-mentioned neural networks for analysis and prediction. Through this approach, we aimed to accurately predict the probability of a gamer continuing on the platform, providing valuable insights as a whole.

9. Community Detection: We use community detection algorithms such as Label Propagation, Louvain, Leiden to perform detection of communities among the gamers. For layout purpose, fruchterman reingold layout was used which is algorithm with purpose of spring layout, optimized for data having a thousands of nodes. The visualization of the whole network is a highly computationally expensive task. Yet in order to get some overview over the distribution we attempt to visualise a smaller graph in form of sub-graph. For comparision and measuring the quality of partition in network, we use Modularity as fitness statistic. We observe the community formation with various resolution values to see

how the modularity varies with resolution and choose best parameter. Along with grouping the nodes into communities on properties of complete graph, we also study graph generated on the segregated properties like language.

4. Experimental setup

Dataset: The dataset used in this project is created by the *Benedek Rozemberczki, Rik Sarkar*in their paper *(Twitch Gamers: a Dataset Evaluating Proximity Preserving and Structural Role-based Node Embeddings)* [6] and made publicly available. The authors have released the clean subset of the data they crawled of the largest connected component of the Twitch social network with snowball sampling. The Twitch network data has the following characteristics:

1. The data is an un-directed and unweighted graph data.
2. The graph has **168,114** nodes.
3. The graph has **6,797,557** edges.
4. The network data has the following 9 node features:
   (a) Identifier: It is the node identifier representing Twitch user.
   (b) Dead Account: It's a categorical variable with values = [0,1] representing whether the user account is inactive or active
   (c) Broadcaster Language: It's a categorical variable with values = ['EN', 'FR', 'KO', 'JA', 'RU', 'PL', 'DE', 'ES', 'IT', 'PT', 'OTHER', 'TR', 'ZH', 'SV', 'NL', 'TH', 'CS', 'DA', 'HU', 'FI', 'NO'] representing languages used for broadcasting.
   (d) Affiliate Status: It's a categorical variable with values = [0,1] indicating if the user is a qualified streamer who can earn an income by live streaming video games.
   (e) Explicit Content: It's a categorical variable with values = [0,1] indicating if the content is explicit.
   (f) Creation Date: It's a date type variable indicating the joining date of the user.
   (g) Last Update: It's a date-type variable indicating the last stream of the user.
   (h) View Count: It's a discrete quantitative variable representing the number of views on the channel.
   (i) Account Lifetime: It's a discrete quantitative variable representing the days between the first and last stream.

Hardware Platforms: Given the size of the dataset, we used the IU shark system named al which is a GPU server with 4 NVIDIA Quadro RTX 6000 GPUs. The system is a dual-socket, 10-core (20 total cores) Intel Xeon system with 256GB of memory.

Software systems: We used Python programming language along with several relevant packages. Specifically, we used the NetworkX library for analyzing the Twitch Gamers social network dataset, and the PyTorch framework for implementing and evaluating graph machine learning models for tasks such as node classification, prediction, and clustering.

Data Preprocessing: In order to run Classification and Link prediction using Graph Neural Networks, The data needs to be in the PyTorch-geometric data format. Therefore, we had to preprocess the data. The data was available to us as two comma-separated files named edges and features. We merged the dataset using the numeric_id of the node as the key.

The Categorical columns like Language are encoded to numerical values from 0 to 20. The date columns created_at and updated_at are separated into date, month, and year columns to be used for the classification. The datatypes of all the columns are changed to int32. The target columns like dead_account, language, affiliate, and mature were stored in a different dataframe and the rest of the data was used accordingly as the input depending on the classification task which we were trying to perform.

The dataset was split into train, test, and validation in the ratio of 85:45:35 out of the whole dataset.

Hyper-parameters: The model hyperparameters such as epochs, learning_rate and weight_decay. For all the models we used learning_rate with 0.01, weight_decay of 5e-4 and 200 epochs. We finalized these hyperparameters by testing the accuracy of the model by running with a set of hyperparameters and observing the changes. However, We observed almost similar results when changing the hyperparameters.

	Hidden Dimension = 2	Hidden Dimension = 4	Hidden Dimension = 8	Hidden Dimension = 16	Hidden Dimension = 64
Accuracy	0.964733	0.969267	0.969399	0.030601	0.968725
Number of Epochs	0.000000	37.000000	13.000000	2.000000	1.000000

Figure 2. Hidden Dimensions

	Number of Layers = 1	Number of Layers = 2	Number of Layers = 4	Number of Layers = 8
Accuracy	0.969399	0.969254	0.969399	0.969399
Number of Epochs	3.000000	2.000000	2.000000	1.000000

Figure 3. Number of Layers

	Learning Rate = 1	Learning Rate = 0.5	Learning Rate = 0.1	Learning Rate = 0.01	Learning Rate = 0.0001
Accuracy	0.969399	0.969399	0.968844	0.967918	0.968923
Number of Epochs	1.000000	0.000000	0.000000	1.000000	42.000000

Figure 4. Learning Rate

5. Results

We observe the following for the Twitch social network:

1. Degree Distribution: Degree distribution helped to understand the structure of a network and how nodes are connected to each other.

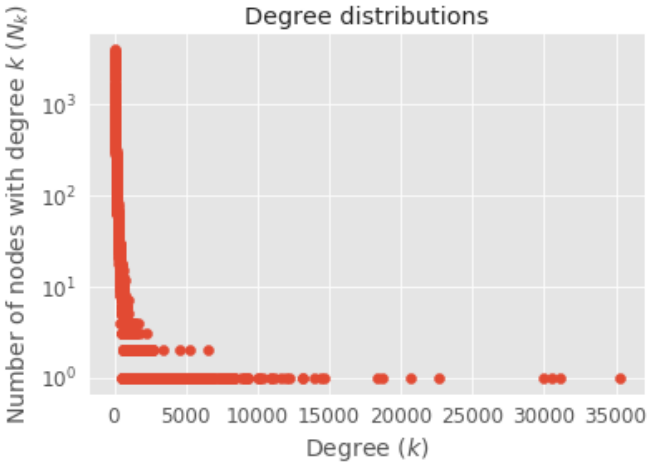

Figure 5. Degree Distribution

The plot indicates a scale-free degree distribution, which tells that there are a few users with an exceptionally large number of followers, while most of the users have relatively few numbers of followers. This implies that the network is characterized by a few highly connected "hubs" that play a critical role in connecting other users in the network.

2. Clustering Coefficient Distribution: Clustering Coefficient Distribution here describes the frequency distribution of clustering coefficients across all nodes in the network.

Figure 6. Clustering Coefficient

The average clustering coefficient is 0.1599 showing on average, a node in the network has a 0.1599 fraction of its neighbors connected to each other and indicating that the nodes are sparsely connected to their neighbors.

3. Shortest Path Distributions: We plot this to understand the network's connectivity and efficiency of communication between nodes.

Figure 7. Shortest Path Distributions

From the plot, we can see that on average, most of the nodes can be reached by utmost 3 hops and maximum 4 hops.

4. Link Prediction - While performing Link prediction, our initial approach was to embed the edges dataset for the Twitch users using embedding algorithms such as Node2Vect and DeepWalk. However, embedding algorithms like Node2Vec and DeepWalk can be computationally expensive and require a lot of memory, especially for large graph datasets like the edges dataset of Twitch streamers.

   This is largely due to the fact that these algorithms need periodically traverse the whole graph in order to create random walks, which may take a very long time for very large graphs. It is possible for the embedding process to be computationally expensive due to a large embedding dimension or a large number of negative samples. In addition, these methods may be sensitive to the choice of hyperparameters, such as window size and walk length, which may be challenging to adjust on big networks.

   Therefore, we opt to use embedding techniques that are more efficient and scalable, like GraphSAGE and GNN, which are better suited for sizable graph datasets. We discovered that these techniques are more suited to real-world networks like the Twitch streamers network because they can efficiently update embeddings and manage huge graphs. Further work on link prediction may involve experimenting with different hidden layer counts, activation functions, and flattening and normalizing techniques to enhance the model's accuracy.

   Upon running the GCN for 100 epochs, we observed a decrease in the loss value with an increase in the number of epochs for both GCN and Graph Sage models as below -

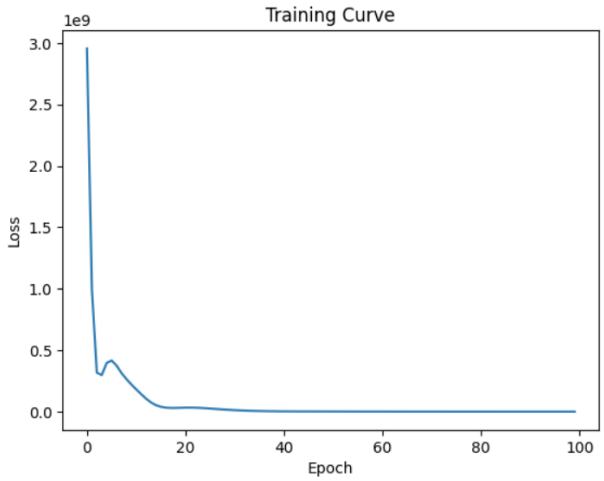

Figure 8. Graph Convolution Network

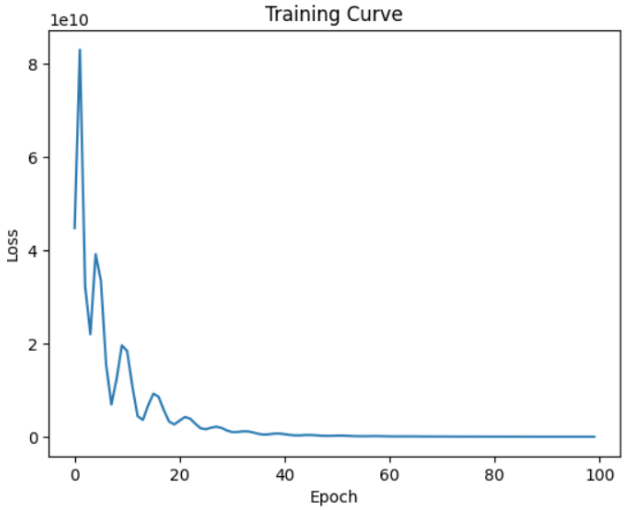

Figure 9. Graph SAGE Network

Upon comparing the accuracy after 100 epochs, we can see that the GCN performs a little better compared to the graph SAGE network as below -

Train Loss on 1st Epoch	Train Loss on 100th Epoch	Final Validation Accuracy	Final Test Accuracy
3895620849.8101	9196.7791	0.5	0.5

Figure 10. Final Result with GCN

Train Loss on 1st Epoch	Train Loss on 100th Epoch	Final Validation Accuracy	Final Test Accuracy
41470287465.3809	21560162.4062	0.4846	0.4929

Figure 11. Final Result with Graph Sage

5. Lifetime Prediction - The GCN and SAGEConv models were trained for over 100 epochs to predict the probability of a Twitch gamer continuing in the future. However, despite the long training period, the accuracy achieved was quite low, at around 19 percent. This can be attributed to several factors, including a possible lack of data. The accompanying figure illustrates the statistics related to this study.

Table 1. Lifetime Prediction Accuracy

Model – 100 Epochs	Accuracy
GCNConv	0.1899
SAGEConv	0.1892

6. Node Classification Results

Table 2. Node Classification Results

Table 3. Dead Account Classification Accuracy

Model – 200 Epochs	Training Accuracy	Testing Accuracy
GCNConv	0.9695	0.9694
SAGEConv	0.9609	0.9694
GATConv	0.9665	0.9694

Table 4. Language Classification Accuracy

Model – 200 Epochs	Training Accuracy	Testing Accuracy
GCNConv	0.5587	0.7402
SAGEConv	0.4223	0.7385
GATConv	0.7254	0.7402

Table 5. Affiliate Status Classification Accuracy

Model – 200 Epochs	Training Accuracy	Testing Accuracy
GCNConv	0.5148	0.5142
SAGEConv	0.4975	0.5267
GATConv	0.5387	0.5151

Table 6. Explicit Content Classification Accuracy

Model – 200 Epochs	Training Accuracy	Testing Accuracy
GCNConv	0.5294	0.5313

7. Community Detection: The viewers and streamers are distinct in the visualization on sub-graph as seen in fig. 12.

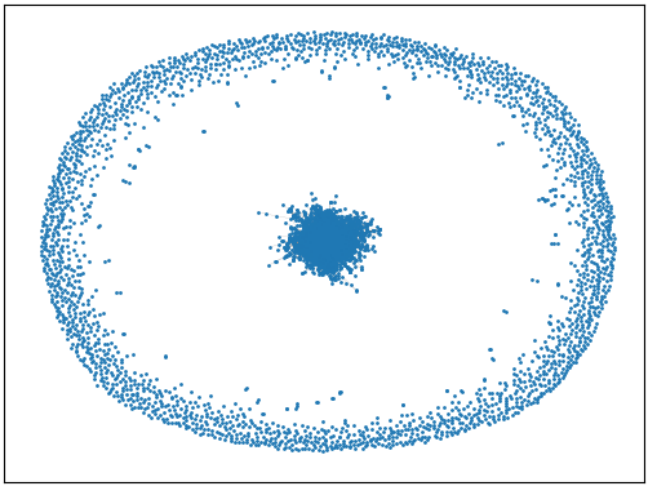

Figure 12. Node Distribution

For community detection, three algorithms were experimented namely Label Propagation, Louvain, and Leiden with Markov having no result due to extreme resource needs. While the modularity scores for label propagation and leiden were very small to guarantee the fitness of method on data, whereas Louvain algorithm showed the highest modularity scores of 0.42 for 1. resolution.

Algorithm	Sub-graph fraction	Number of clusters	Modularity
Label propagation	0.05	2483	0.039
Louvain	0.05	2700	0.55
Leiden	0.05	2697	0.56
Label propagation	1	79	0.035
Louvain	1	20	0.424
Leiden	1	19	-0.00014
Markov clustering	1	Memory insufficient	Memory insufficient

Figure 13. Community Detection Results

We also performed community detection specifically on graphs with languages that showed the best modularity of 0.425

6. Conclusions

1. The analysis of the Twitch Gamer Social Network was performed using Networkx and the characteristics are as follows: It is a Scale Free Network where nodes are sparsely connected to the neighbors.
2. The Link Prediction was performed on the dataset and we observed better accuracy using GCN compared to GraphSAGE. However, the accuracy of the prediction stayed at approximately 50% even after tuning the model hyperparameters.
3. The Node Classification task was performed on the dataset using the node features such as dead_account, mature, affiliate, and language. The accuracy of dead_account classification turned out to be 96%, the accuracy of language classification was observed to be 74%, affiliate status classification accuracy was reported as 52%, and explicit content classification gave the accuracy of 53%.
4. Community detection task was performed on the entire network as well as the languages-only network with the highest modularity scores of 0.422 and 0.425 respectively

Team member contributions

Jayesh Prasad Anandan: Data Preprocessing and Node Classification - Dead Account Classification, Language Classification, Affiliation Status Classification, and Explicit Content Classification using GCN, GraphSAGE, and GAT.

Shivani Pal: Analysis of the network - degree distribution of network, clustering coefficient distribution, shortest path distribution, and explicit content classification using GCN-Conv.

Bhakti Patrawala: Link prediction using Convolution neural networks and graph sage network.

Himani Ravindra Borana: Lifetime Prediction using Graph Convolutional Network (GCN) and SAGEConv model.

Laxmikant Maheshji Kabra: Community detection and comparison between various parameters.

References

[1] *Thomas N. Kipf, Max Welling. "Semi-Supervised Classification with Graph Convolutional Networks" arXiv:1609.02907v4 (2017).*

[2] *William L. Hamilton, Rex Ying, Jure Leskovec. "Inductive Representation Learning on Large Graphs" arXiv preprint arXiv:1706.02216v4 (2018).*

[3] *Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, Yoshua Bengio. "Graph Attention Networks" arXiv:1710.10903v3 (2018).*

[4] *Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, Qiaozhu Mei. "LINE: Large-scale Information Network Embedding." arXiv preprint arXiv:1503.03578v1 (2015).*

[5] *Alexey Dosovitskiy, Philipp Fischer, Jost Tobias Springenberg, Martin Riedmiller, Thomas Brox. "Discriminative Unsupervised Feature Learning with Exemplar Convolutional Neural Networks" arXiv preprint arXiv:1406.6909v2 (2015).*

[6] *Rozemberczki, Benedek, and Rik Sarkar. "Twitch gamers: a dataset for evaluating proximity preserving and structural role-based node embeddings." arXiv preprint arXiv:2101.03091 (2021).*