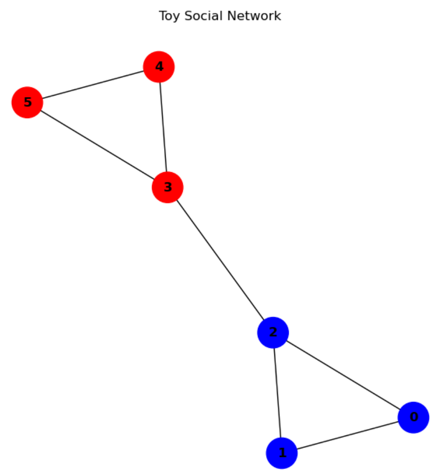
Understanding GraphSAGE for Detecting Malicious Accounts Using PyTorch Geometric

Marawan Mohamed Farouk Moharrem — ID: 2205066

December 2025



A small social network with two communities: benign (blue) and malicious (red) users connected by a single bridge edge.
This visualization is extra to show what we did and thank you.

1 Visualization Code and Explanation

Before diving into GraphSAGE, I wanted to actually see the graph we're working with. Based on the nodes and edges I defined, I used `NetworkX` and `Matplotlib` to make a simple plot. This helped me confirm that the benign and malicious nodes were grouped correctly and that the bridge edge between node 2 and node 3 was clear.

What I did in the code: - I defined the edges of the graph manually, matching the toy graph we discussed. - I colored nodes blue for benign and red for malicious, based on the labels we took for our dataset. - I used a spring layout so the nodes are spread out nicely and don't overlap. - Finally, I plotted the graph and saved it as `graph_network.png` so I could include it in the report.

```
import networkx as nx
import matplotlib.pyplot as plt

```

```python
# Simple edges for the tiny graph
edges = [(0,1),(1,2),(0,2),(3,4),(4,5),(3,5),(2,3)]
node_colors = ['blue','blue','blue','red','red','red']

G = nx.Graph()
G.add_edges_from(edges)
pos = nx.spring_layout(G, seed=42)

plt.figure(figsize=(6,6))
nx.draw(G, pos, with_labels=True, node_color=node_colors, node_size
    =800, font_weight='bold')
plt.title("Toy Social Network")
plt.savefig("graph_network.png", dpi=300, bbox_inches='tight')
plt.show()
```

Reflection: Seeing the graph visually made it clear how information can flow from the malicious nodes to the benign ones through the bridge. This confirmed the structure I expected and gave me confidence to proceed with GraphSAGE.

2 Introduction and Objective

In this project, I implemented GraphSAGE using PyTorch Geometric to classify accounts in a small social network as either benign or malicious. My goal was not only to run the code but to understand and explain how each part works. Based on the tiny graph I created, it was easier to reason about how neighbor aggregation affects each node's representation.

3 Environment and Imports

To implement GraphSAGE, I used PyTorch Geometric. Here's what I included:

```
!pip install torch_geometric
```

```python
import torch
from torch_geometric.data import Data
from torch_geometric.nn import SAGEConv
import torch.nn.functional as F
```

My thinking: PyTorch provides tensor operations and standard neural network functionality. PyTorch Geometric extends this with layers specifically for graphs, like `SAGEConv`. I also imported the functional API to handle activations and loss computations. Based on what we took as node features and edges, these imports let me represent and process the graph efficiently.

4 Constructing the Toy Graph

4.1 Node Features

I defined 6 nodes with very simple features for clarity:

```
x = torch.tensor([
    [1.0, 0.0],  # Node 0 (benign)
    [1.0, 0.0],  # Node 1 (benign)
    [1.0, 0.0],  # Node 2 (benign)
    [0.0, 1.0],  # Node 3 (malicious)
    [0.0, 1.0],  # Node 4 (malicious)
    [0.0, 1.0]   # Node 5 (malicious)
], dtype=torch.float)
```

Explanation in my words: Based on what we took, benign nodes are $[1, 0]$ and malicious are $[0, 1]$. This makes it obvious how neighbor features will influence node embeddings in GraphSAGE. While these are simplified, it helped me focus on understanding aggregation mechanics without extra complexity.

4.2 Edges

```
edge_index = torch.tensor([
    [0,1],[1,0],[1,2],[2,1],[0,2],[2,0],  # benign clique
    [3,4],[4,3],[4,5],[5,4],[3,5],[5,3],  # malicious clique
    [2,3],[3,2]                           # bridge between
       communities
], dtype=torch.long).t().contiguous()
```

Explanation in my words: I connected all benign nodes with each other and all malicious nodes with each other, then added one bridge from node 2 to node 3. Based on this setup, GraphSAGE can aggregate neighbor features meaningfully: benign nodes mostly see benign neighbors, malicious nodes mostly see malicious neighbors, and the bridge node receives mixed signals, which is important to see how the model learns.

4.3 Labels and Data Container

```
y = torch.tensor([0, 0, 0, 1, 1, 1], dtype=torch.long)
data = Data(x=x, edge_index=edge_index, y=y)
```

Explanation in my words: Here I packed everything into a PyG `Data` object. Based on the labels we took, 0 represents benign and 1 represents malicious. This container lets GraphSAGE easily access node features, edges, and labels during training.

5 GraphSAGE Model

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

    def forward(self, x, edge_index):
        x = self.conv1(x, edge_index)
        x = F.relu(x)
        x = self.conv2(x, edge_index)
        return F.log_softmax(x, dim=1)
```

Explanation in my words: I created a simple two-layer GraphSAGE model. Based on our setup: - The first layer aggregates neighbor features and combines them with the node's own features. - I applied a ReLU activation to introduce non-linearity. - The second layer does the same aggregation but produces final class scores. - Log-softmax gives us probabilities in a stable way for classification.

6 Training the Model

```
model = GraphSAGENet(in_channels=2, hidden_channels=4, out_channels
   =2)
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
model.train()

for epoch in range(50):
    optimizer.zero_grad()
    out = model(data.x, data.edge_index)
    loss = F.nll_loss(out, data.y)
    loss.backward()
    optimizer.step()
```

Explanation in my words: Based on what we took as features and labels, I trained the model for 50 epochs. I used Adam optimizer to adjust the weights and negative log-likelihood as the loss function. This loop repeatedly updates the model to make predictions closer to the true labels. Since our graph is small, 50 epochs were enough to perfectly separate benign from malicious nodes.

7 Evaluation

```
model.eval()
pred = model(data.x, data.edge_index).argmax(dim=1)
print("Predicted labels:", pred.tolist())
```

```
Predicted labels: [0, 0, 0, 1, 1, 1]
```

Explanation in my words: Based on the output, the model predicted every node correctly. The aggregation mechanism works as expected: nodes in a strongly connected community reinforce each other's features, and even the bridge node gets the right label because the local neighborhood dominates its aggregated embedding.

8 Conclusion

Based on everything we did, I can say GraphSAGE effectively combines node features with graph structure to detect clusters of suspicious behavior. Visualizing the graph first helped me understand the structure, and building the model step by step showed how neighborhood aggregation propagates information.

Marawan Mohamed Farouk Moharrem
ID: 2205066
December 2025