**Ahmed Nabil Nour Ahmed** **2205245**

# Explanation of GraphSAGE on a Toy Social Network (6 Users)

Demonstrate how a simple **2-layer GraphSAGE** model can classify users in a small social network as either **benign (0)** or **malicious (1)** by leveraging both:

- Node features
- Graph structure (friendship connections)

This is a classic example of **Graph Neural Network (GNN)** for node classification.

---

## 1. Import Libraries

!pip install torch_geometric

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

- torch: Core PyTorch library for tensors and neural networks.
- torch_geometric: Extension for graph deep learning (PyG).
- SAGEConv: The GraphSAGE convolution layer (key building block).
- F: Shortcut for common functions (ReLU, log_softmax, etc.).

---

## 2. Define Node Features (x)

```
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)
```

- Each node has a 2-dimensional feature vector.
- The first 3 nodes have low values  these represent **benign** users
- The last 3 nodes have higher values  these represen **malicious** users
- Benign users: [1, 0] → like a "clean profile" signal.
- Malicious users: [0, 1] → like a "suspicious profile" signal.
- This is a simplified but realistic setup: in real fraud detection, features could be behavior stats, account age, etc.

---

## 3. Define Graph Structure (Edges)

Python

```
edge_index = torch.tensor([
    [0,1], [1,0], [1,2], [2,1], [0,2], [2,0],  # Benign clique (0-1-2 fully connected)
    [3,4], [4,3], [4,5], [5,4], [3,5], [5,3],  # Malicious clique (3-4-5 fully connected)
    [2,3], [3,2]                                # One bridge edge between groups
], dtype=torch.long).t().contiguous()
```

we create edge_index, which tells the model **who is connected to who** in the graph

Nodes 0, 1, and 2 are connected together  this forms a benign group

Nodes 3, 4, and 5 are connected together  this forms a malicious group

---

## 4. Node Labels

Python

```
y = torch.tensor([0, 0, 0, 1, 1, 1], dtype=torch.long)
```

- Ground truth: first 3 nodes = benign (0), last 3 = malicious (1).

---

## 5. Create PyG Data Object

Python

```
data = Data(x=x, edge_index=edge_index, y=y)
```

- Packs everything into a single object that PyG models expect.

---

## 6.Building the GraphSAGE Model

Python

```python
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):
        # Layer 1: Aggregate 1-hop neighbors
        x = self.conv1(x, edge_index)
        x = F.relu(x)                    # Non-linearity

        # Layer 2: Aggregate again (now sees 2-hop neighborhood)
        x = self.conv2(x, edge_index)

        return F.log_softmax(x, dim=1)    # Log probabilities for each class
```

- **GraphSAGE mechanism**: Each node updates its representation by aggregating features from its neighbors.
- After two layers → each node sees information from nodes up to **2 hops away**.
- Due to the bridge edge (2–3), information can flow between the two communities.
  creates a neural network with two GraphSAGE layers
  **Layer 1:**
  self.conv1 = SAGEConv(2, 4)
  Takes 2 input features
  Produces 4 new features
  **Layer 2:**
  self.conv2 = SAGEConv(4, 2)
  Takes the 4 features.
  Outputs 2 values, one for each class: benign or malicious

---

## 7. Model Instantiation

```python
model = GraphSAGENet(in_channels=2, hidden_channels=4, out_channels=2)
```

- Input: 2 features
- Hidden layer: 4 dimensions
- Output: 2 classes (benign vs malicious)

---

## 8. Training Setup

Python

```
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)    # Negative log-likelihood loss
    loss.backward()
    optimizer.step()
```

- Full-batch training (all 6 nodes at once — fine for toy graph).
- The model learns to predict correct labels by minimizing classification error.

---

## 9. Final Predictions

Python

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

**Typical output after training:**

text

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

**100% accuracy**