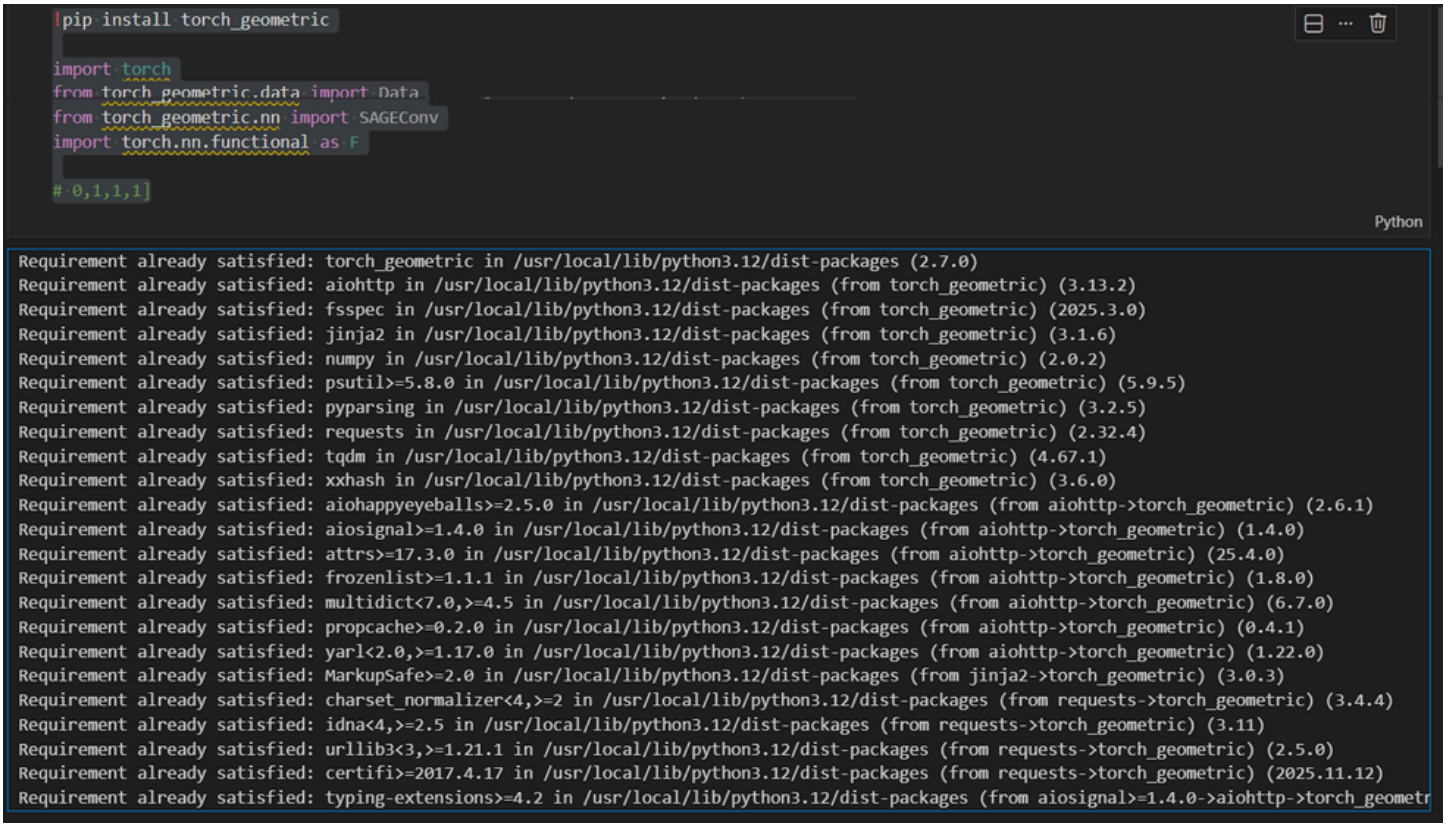
Meral Ayman Abbas

2205116

section 9 report



```
!pip install torch_geometric

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

# 0,1,1,1]
```

Python

```
Requirement already satisfied: torch_geometric in /usr/local/lib/python3.12/dist-packages (2.7.0)
Requirement already satisfied: aiohttp in /usr/local/lib/python3.12/dist-packages (from torch_geometric) (3.13.2)
Requirement already satisfied: fsspec in /usr/local/lib/python3.12/dist-packages (from torch_geometric) (2025.3.0)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.12/dist-packages (from torch_geometric) (3.1.6)
Requirement already satisfied: numpy in /usr/local/lib/python3.12/dist-packages (from torch_geometric) (2.0.2)
Requirement already satisfied: psutil>=5.8.0 in /usr/local/lib/python3.12/dist-packages (from torch_geometric) (5.9.5)
Requirement already satisfied: pyparsing in /usr/local/lib/python3.12/dist-packages (from torch_geometric) (3.2.5)
Requirement already satisfied: requests in /usr/local/lib/python3.12/dist-packages (from torch_geometric) (2.32.4)
Requirement already satisfied: tqdm in /usr/local/lib/python3.12/dist-packages (from torch_geometric) (4.67.1)
Requirement already satisfied: xxhash in /usr/local/lib/python3.12/dist-packages (from torch_geometric) (3.6.0)
Requirement already satisfied: aiohappyeyeballs>=2.5.0 in /usr/local/lib/python3.12/dist-packages (from aiohttp->torch_geometric) (2.6.1)
Requirement already satisfied: aiosignal>=1.4.0 in /usr/local/lib/python3.12/dist-packages (from aiohttp->torch_geometric) (1.4.0)
Requirement already satisfied: attrs>=17.3.0 in /usr/local/lib/python3.12/dist-packages (from aiohttp->torch_geometric) (25.4.0)
Requirement already satisfied: frozenlist>=1.1.1 in /usr/local/lib/python3.12/dist-packages (from aiohttp->torch_geometric) (1.8.0)
Requirement already satisfied: multidict<7.0,>=4.5 in /usr/local/lib/python3.12/dist-packages (from aiohttp->torch_geometric) (6.7.0)
Requirement already satisfied: propcache>=0.2.0 in /usr/local/lib/python3.12/dist-packages (from aiohttp->torch_geometric) (0.4.1)
Requirement already satisfied: yarl<2.0,>=1.17.0 in /usr/local/lib/python3.12/dist-packages (from aiohttp->torch_geometric) (1.22.0)
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.12/dist-packages (from jinja2->torch_geometric) (3.0.3)
Requirement already satisfied: charset_normalizer<4,>=2 in /usr/local/lib/python3.12/dist-packages (from requests->torch_geometric) (3.4.4)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.12/dist-packages (from requests->torch_geometric) (3.11)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.12/dist-packages (from requests->torch_geometric) (2.5.0)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.12/dist-packages (from requests->torch_geometric) (2025.11.12)
Requirement already satisfied: typing-extensions>=4.2 in /usr/local/lib/python3.12/dist-packages (from aiosignal>=1.4.0->aiohttp->torch_geometr
```

The first part just shows that all the needed packages were already installed.

The second part is importing the tools I need to work with graph neural networks in PyTorch Geometric.

import torch : I'm loading the main PyTorch library.

from torch_geometric.data import Data : I'm loading a class that helps me store graph data.

from torch_geometric.nn import SAGEConv : I'm loading the GraphSAGE layer, which is a type of neural network layer made for graph data.

import torch.nn.functional as F : I'm loading a bunch of helpful functions used inside neural networks.

```python
#--- Define a small graph with 6 nodes ---
# Node features (2 features per node).
# Here benign users have [1, 0] and malicious have [0, 1] for illustration.
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,
)
```

creating the features for a small graph that has 6 nodes.
Each node has 2 numbers (two features).

Benign (normal) users :[1.0, 0.0]

Malicious users :[0.0, 1.0]

Nodes 0, 1, and 2 are benign, so they get [1, 0]

Nodes 3, 4, and 5 are malicious, so they get [0, 1]

```python
# Edge list (undirected). Connect benign users (0-1-2 fully connected)
# and malicious users (3-4-5 fully connected), plus one cross-edge 2-3.
edge_index = (
    torch.tensor(
        [
            [0, 1],
            [1, 0],
            [1, 2],
            [2, 1],
            [0, 2],
            [2, 0],
            [3, 4],
            [4, 3],
            [4, 5],
            [5, 4],
            [3, 5],
            [5, 3],
            [2, 3],
            [3, 2],  # one connection between a benign (2) and malicious (3)
        ],
        dtype=torch.long,
    )
    .t()
    .contiguous()
)
```

creating the edges of the graph
Edges tell us which nodes are connected to each other.

Because the graph is undirected every connection is written twice:

if node 0 is connected to node 1, we write [0,1] and [1,0]

-benign users know each other.

-malicious users are all connected.

-There is one connection between:

node 2 (benign)

node 3 (malicious)

```python
# Labels: 0 = benign, 1 = malicious
# y contains the true labels of the 6 nodes:
# Nodes 0, 1, 2 are benign → label 0
# Nodes 3, 4, 5 are malicious → label
# data is a torch_geometric.data.Data object containing
# x: node features
# edge_index: graph connections (edges)
# y: labels
y = torch.tensor([0, 0, 0, 1, 1, 1], dtype=torch.long)

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


# --- Define a two-layer GraphSAGE model ---
# his defines a 2-layer GraphSAGE neural network.
# in_channels=2 means each node has 2 features.
# hidden_channels=4 creates a 4-dimensional hidden embedding.
# out_channels=2 means the model outputs scores for 2 classes (benign and malicious).

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

    def forward(self, x, edge_index):
        # First layer: sample neighbors and aggregate
        x = self.conv1(x, edge_index)
        x = F.relu(x)  # non-linear activation
        # Second layer: produce final embeddings/class scores
        x = self.conv2(x, edge_index)
        return F.log_softmax(x, dim=1)  # log-probabilities for classes
```

creates the true class labels for the 6 nodes.

Nodes 0, 1, 2 → benign → label 0 Nodes 3, 4, 5 → malicious → label 1

y tells the model what each node really is so it can learn

creates one graph that contains:

x: the node features

edge_index: the connections between nodes

y: the labels we want the model to learn

then a two-layer GraphSAGE model that:

Reads the graph

Learns from neighbors

Predicts whether each node is benign or malicious

```
# Instantiate model: input dim=2, hidden=4, output dim=2 (benign vs malicious)
model = GraphSAGENet(in_channels=2, hidden_channels=4, out_channels=2)

# Simple training loop
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.backward()
    optimizer.step()
```

builds GraphSAGE network with:

2 input features per node

4 hidden units

2 output classes (0 = benign, 1 = malicious)

optimizer updates the model's weights to make it learn.

Adam is a common optimizer

lr=0.01 is the learning rate

trained the model for 50 epochs

In each epoch:

Clear previous gradients

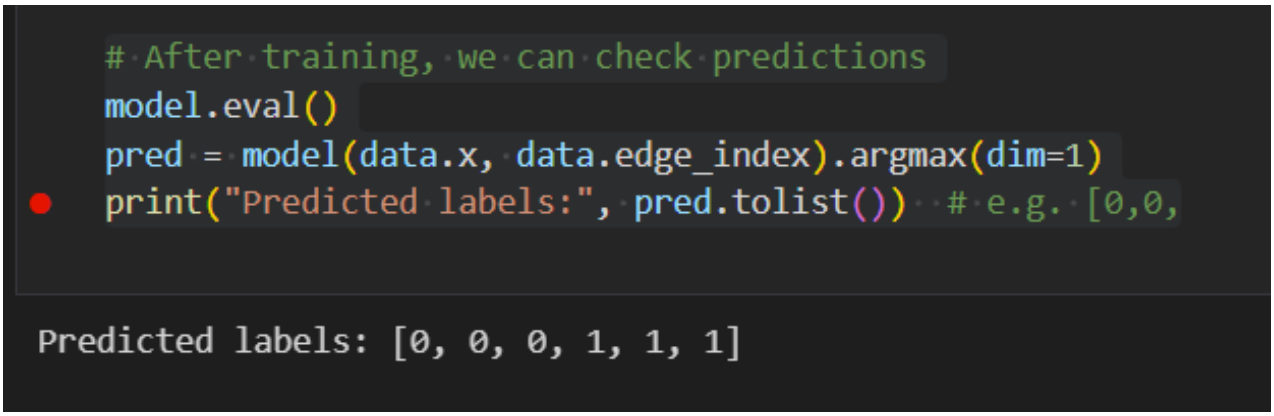
Run the model to get predictions

Compute loss using negative log-likelihood

Backpropagate to compute gradients

Update model weights

The model gets better every epoch at predicting which nodes are benign or malicious.



Switch model to evaluation mode.

Feed it the graph again to get predictions.

Use argmax to choose the class with the highest score for each node.

Print the predicted labels.