Python code

"""

Detecting Emerging Social Movements Using a simple EvolveGCN-style DGNN

- Synthetic temporal graph + text generator included

- Lightweight DGNN implemented in PyTorch (no torch\_geometric required)

- Binary classifier: whether a time-slice contains an emerging movement

- Metrics: accuracy, precision, recall, f1, and a simple lead-time proxy

Author: ChatGPT (example implementation)

Date: 2025

"""

import math

import random

from typing import List, Tuple, Dict

import numpy as np

import torch

import torch.nn as nn

import torch.optim as optim

from sklearn.metrics import accuracy\_score, precision\_recall\_fscore\_support

from tqdm import trange, tqdm

import warnings

warnings.filterwarnings("ignore")

# ---------------------------

# Utilities & Synthetic Data

# ---------------------------

def set\_seed(seed=42):

random.seed(seed)

np.random.seed(seed)

torch.manual\_seed(seed)

set\_seed(42)

def generate\_synthetic\_temporal\_data(

num\_days: int = 60,

base\_users: int = 500,

burst\_users: int = 150,

avg\_edges\_per\_day: int = 800,

burst\_day\_indices: List[int] = None,

vocab\_size: int = 2000,

text\_dim: int = 128

) -> Tuple[List[Dict], List[int]]:

"""

Generate synthetic temporal graphs and per-node text embeddings.

Returns:

- graphs: list of dicts with keys { 'nodes': [uids], 'edges': [(u,v)], 'text\_feats': {uid:vec} }

- labels: list of 0/1 indicating if 'emerging movement' in that day

Notes:

- burst\_day\_indices: days where an emerging movement starts.

- This synthetic dataset mimics increased interactions among a subset of users

and semantic drift in their text embeddings during burst days.

"""

if burst\_day\_indices is None:

# put 2 bursts in the timeline

burst\_day\_indices = [15, 40]

graphs = []

labels = []

base\_user\_ids = list(range(base\_users))

# random base embeddings for each user

user\_base\_text = {u: np.random.normal(scale=1.0, size=(text\_dim,)) for u in base\_user\_ids}

for day in range(num\_days):

nodes = base\_user\_ids.copy()

edges = []

text\_feats = {}

# normal interactions

for \_ in range(avg\_edges\_per\_day):

u = random.choice(nodes)

v = random.choice(nodes)

if u != v:

edges.append((u, v))

label = 0

# if this day is a burst day: create a cohesive burst community

for b in burst\_day\_indices:

if day >= b and day < b + 3: # short early phase lasts 3 days

label = 1

# create burst\_users new edges among a subset

community = random.sample(nodes, burst\_users)

for \_ in range(burst\_users \* 2):

u = random.choice(community)

v = random.choice(community)

if u != v:

edges.append((u, v))

# perturb text embeddings for community (semantic coherence)

community\_shift = np.random.normal(loc=1.0, scale=0.5, size=(text\_dim,))

for u in community:

text\_feats[u] = user\_base\_text[u] + community\_shift

# fill any missing text embeddings with base ones and add small noise

for u in nodes:

if u not in text\_feats:

text\_feats[u] = user\_base\_text[u] + np.random.normal(scale=0.01, size=(text\_dim,))

graphs.append({'nodes': nodes, 'edges': edges, 'text\_feats': text\_feats})

labels.append(label)

return graphs, labels

# ---------------------------

# Simple GCN layer (matrix)

# ---------------------------

class SimpleGCNLayer(nn.Module):

def \_\_init\_\_(self, in\_dim, out\_dim, bias=True):

super().\_\_init\_\_()

self.linear = nn.Linear(in\_dim, out\_dim, bias=bias)

def forward(self, X, A\_norm):

# X: [N, in\_dim], A\_norm: [N, N] normalized adjacency

return torch.relu(self.linear(A\_norm @ X))

# ---------------------------

# EvolveGCN-like DGNN

# ---------------------------

class EvolveGCN\_O(nn.Module):

"""

EvolveGCN-O style: evolve the weights of a GCN layer via a GRU that

takes a summary of past node embeddings as input.

This implementation uses matrix parameters W that are evolved per timestep.

"""

def \_\_init\_\_(self, in\_dim, hid\_dim, out\_dim, gru\_hidden\_dim=128, device='cpu'):

super().\_\_init\_\_()

self.device = device

self.in\_dim = in\_dim

self.hid\_dim = hid\_dim

self.out\_dim = out\_dim

# initialize weight matrices as parameters (will be evolved)

# For simplicity implement two-layer GCN whose weights we evolve

self.W1 = nn.Parameter(torch.randn(in\_dim, hid\_dim) \* 0.1)

self.W2 = nn.Parameter(torch.randn(hid\_dim, out\_dim) \* 0.1)

# GRU to evolve flattened weight matrices

self.gru1 = nn.GRUCell(in\_dim, gru\_hidden\_dim)

self.gru2 = nn.GRUCell(hid\_dim, gru\_hidden\_dim)

# small MLP to map GRU hidden state to weight updates

self.w1\_mapper = nn.Linear(gru\_hidden\_dim, in\_dim \* hid\_dim)

self.w2\_mapper = nn.Linear(gru\_hidden\_dim, hid\_dim \* out\_dim)

# final classifier on node embeddings (pooled)

self.classifier = nn.Sequential(

nn.Linear(out\_dim, out\_dim//2),

nn.ReLU(),

nn.Linear(out\_dim//2, 1),

nn.Sigmoid()

)

# GRU hidden states

self.h1 = None

self.h2 = None

def reset\_state(self, batch\_size=None):

# initialize GRU hidden states to zeros

# batch\_size not used; keep hidden dim consistent

device = self.device

self.h1 = torch.zeros(self.gru1.hidden\_size, device=device)

self.h2 = torch.zeros(self.gru2.hidden\_size, device=device)

def forward(self, X, A\_norm):

"""

X: [N, in\_dim]

A\_norm: [N, N] normalized adjacency matrix

Returns: node embeddings [N, out\_dim]

"""

# summarize node features for weight evolution (mean)

summary1 = torch.mean(X, dim=0) # [in\_dim]

self.h1 = self.gru1(summary1.unsqueeze(0), self.h1.unsqueeze(0)).squeeze(0)

delta\_w1 = self.w1\_mapper(self.h1) # maps to flattened weights

W1\_evolved = (self.W1 + delta\_w1.view(self.in\_dim, self.hid\_dim)).to(self.device)

# first GCN layer (using evolved W1)

H1 = torch.relu(A\_norm @ (X @ W1\_evolved)) # [N, hid\_dim]

summary2 = torch.mean(H1, dim=0) # [hid\_dim]

self.h2 = self.gru2(summary2.unsqueeze(0), self.h2.unsqueeze(0)).squeeze(0)

delta\_w2 = self.w2\_mapper(self.h2)

W2\_evolved = (self.W2 + delta\_w2.view(self.hid\_dim, self.out\_dim)).to(self.device)

# second GCN layer

H2 = torch.relu(A\_norm @ (H1 @ W2\_evolved)) # [N, out\_dim]

return H2

def predict\_graph(self, node\_embeddings, graph\_nodes\_mask=None):

"""

Produce a graph-level emergence score by mean-pooling node embeddings

and passing to classifier.

"""

if graph\_nodes\_mask is None:

pooled = node\_embeddings.mean(dim=0) # [out\_dim]

else:

pooled = node\_embeddings[graph\_nodes\_mask].mean(dim=0)

score = self.classifier(pooled) # scalar [1]

return score.squeeze()

# ---------------------------

# Helpers: adjacency normalization

# ---------------------------

def build\_adjacency\_matrix(num\_nodes: int, edges: List[Tuple[int,int]], device='cpu'):

A = torch.zeros((num\_nodes, num\_nodes), dtype=torch.float32, device=device)

for (u, v) in edges:

A[u, v] += 1.0

A[v, u] += 1.0 # make it undirected for this demo

# add self-loops

for i in range(num\_nodes):

A[i, i] += 1.0

# normalize: D^{-1/2} A D^{-1/2}

deg = A.sum(dim=1)

D\_inv\_sqrt = torch.diag(1.0 / torch.sqrt(deg))

A\_norm = D\_inv\_sqrt @ A @ D\_inv\_sqrt

return A\_norm

# ---------------------------

# Training & Evaluation

# ---------------------------

def train\_model(graphs, labels, in\_text\_dim=128, hid\_dim=128, out\_dim=64,

epochs=30, lr=1e-3, device='cpu'):

device = torch.device(device)

model = EvolveGCN\_O(in\_dim=in\_text\_dim+2, hid\_dim=hid\_dim, out\_dim=out\_dim, device=device).to(device)

model.reset\_state()

optimizer = optim.Adam(model.parameters(), lr=lr)

bce = nn.BCELoss()

losses = []

for epoch in range(epochs):

model.train()

epoch\_losses = []

for t, G in enumerate(graphs):

nodes = G['nodes']

num\_nodes = len(nodes)

edges = G['edges']

text\_feats = G['text\_feats']

# node features: text vector + simple network features (degree, clustering placeholder)

X = []

for u in nodes:

txt = text\_feats[u]

deg = sum(1 for (a,b) in edges if a==u or b==u)

X.append(np.concatenate([txt, np.array([deg, 0.0])])) # add second network feature placeholder

X = torch.tensor(np.stack(X), dtype=torch.float32, device=device)

# normalized adjacency

A\_norm = build\_adjacency\_matrix(num\_nodes, edges, device=device)

# forward (DGNN evolves at each timestep)

node\_emb = model(X, A\_norm) # [N, out\_dim]

# graph-level prediction

score = model.predict\_graph(node\_emb) # scalar in [0,1]

y\_true = torch.tensor([labels[t]], dtype=torch.float32, device=device)

loss = bce(score.unsqueeze(0), y\_true)

optimizer.zero\_grad()

loss.backward()

optimizer.step()

epoch\_losses.append(loss.item())

avg\_loss = float(np.mean(epoch\_losses))

losses.append(avg\_loss)

if (epoch + 1) % 5 == 0 or epoch==0:

print(f"Epoch {epoch+1}/{epochs} - loss: {avg\_loss:.4f}")

return model, losses

def evaluate\_model(model, graphs, labels, device='cpu'):

model.eval()

device = torch.device(device)

y\_pred = []

y\_true = []

detection\_times = [] # record day indices where model predicts > 0.5

with torch.no\_grad():

for t, G in enumerate(graphs):

nodes = G['nodes']

num\_nodes = len(nodes)

edges = G['edges']

text\_feats = G['text\_feats']

X = []

for u in nodes:

txt = text\_feats[u]

deg = sum(1 for (a,b) in edges if a==u or b==u)

X.append(np.concatenate([txt, np.array([deg, 0.0])]))

X = torch.tensor(np.stack(X), dtype=torch.float32, device=device)

A\_norm = build\_adjacency\_matrix(num\_nodes, edges, device=device)

node\_emb = model(X, A\_norm)

score = model.predict\_graph(node\_emb).item()

pred = 1 if score >= 0.5 else 0

y\_pred.append(pred)

y\_true.append(labels[t])

if pred==1:

detection\_times.append(t)

acc = accuracy\_score(y\_true, y\_pred)

prec, rec, f1, \_ = precision\_recall\_fscore\_support(y\_true, y\_pred, average='binary', zero\_division=0)

# Lead-time proxy: measure average distance between first predicted day and first true positive day

# For synthetic case, we compute: for each true burst, find earliest predicted day >= start-2 and compute offset.

lead\_times = []

true\_bursts = []

# infer bursts by consecutive labels==1 windows start indices

for i in range(len(labels)):

if labels[i]==1 and (i==0 or labels[i-1]==0):

true\_bursts.append(i)

for b in true\_bursts:

# earliest detection >= b-3 (some allowance)

detection = None

for d in detection\_times:

if d >= max(0, b-3):

detection = d

break

if detection is None:

# not detected

lead\_times.append(None)

else:

# positive lead time if detection before the canonical peak (we assume peak at b+1)

lead\_times.append(b - detection) # positive means detected earlier

# compute average lead (ignoring Nones)

numeric = [lt for lt in lead\_times if lt is not None]

avg\_lead = float(np.mean(numeric)) if len(numeric)>0 else None

return {'accuracy': acc, 'precision': prec, 'recall': rec, 'f1': f1, 'lead\_time\_days': avg\_lead}

# ---------------------------

# Example main usage

# ---------------------------

if \_\_name\_\_ == "\_\_main\_\_":

print("Generating synthetic temporal dataset...")

graphs, labels = generate\_synthetic\_temporal\_data(

num\_days=60,

base\_users=400,

burst\_users=100,

avg\_edges\_per\_day=700,

burst\_day\_indices=[12, 34],

vocab\_size=2000,

text\_dim=128

)

print(f"Generated {len(graphs)} daily graphs. Example labels (first 30 days): {labels[:30]}")

device = 'cpu' # change to 'cuda' if GPU available

print("Training DGNN model...")

model, losses = train\_model(graphs, labels, in\_text\_dim=128, hid\_dim=128, out\_dim=64, epochs=30, lr=1e-3, device=device)

print("Evaluating model...")

metrics = evaluate\_model(model, graphs, labels, device=device)

print("Evaluation metrics:")

for k,v in metrics.items():

print(f" {k}: {v}")

# Example: show predicted days

model.eval()

preds = []

with torch.no\_grad():

for t, G in enumerate(graphs):

nodes = G['nodes']

num\_nodes = len(nodes)

edges = G['edges']

text\_feats = G['text\_feats']

X = torch.tensor(np.stack([np.concatenate([text\_feats[u], np.array([sum(1 for (a,b) in edges if a==u or b==u), 0.0])]) for u in nodes]), dtype=torch.float32)

A\_norm = build\_adjacency\_matrix(num\_nodes, edges, device='cpu')

out = model(X, A\_norm)

score = model.predict\_graph(out).item()

preds.append(score)

# print first 30 day predictions

print("First 30 days: predictions (probabilities):")

print(np.round(preds[:30], 3))