```python
"""
radicalization_detection.py
End-to-end pipeline for detecting hate speech and radicalized communities.
Author: V. Sai Surya Laxmi
Date: 2025-10-15
"""

import os
import re
import json
import random
from collections import defaultdict
from typing import List, Dict, Tuple

import numpy as np
import pandas as pd

# NLP / ML
import torch
from torch.utils.data import Dataset, DataLoader
from torch.nn import CrossEntropyLoss
from transformers import BertTokenizerFast, BertForSequenceClassification, AdamW,
get_linear_schedule_with_warmup

# Preprocessing
import spacy
```

```
import nltk
from nltk.corpus import stopwords

# Graph
import networkx as nx
import community as community_louvain  # python-louvain

# Visualization
import matplotlib.pyplot as plt

# Make sure required NLTK data is available
nltk.download('stopwords')

# ----------------------------
# Configuration
# ----------------------------
DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")
MODEL_NAME = "bert-base-uncased"      # or 'distilbert-base-uncased' for faster tests
MAX_LEN = 128
BATCH_SIZE = 16
EPOCHS = 3
LR = 2e-5
RANDOM_SEED = 42

random.seed(RANDOM_SEED)
np.random.seed(RANDOM_SEED)
```

```
torch.manual_seed(RANDOM_SEED)


nlp = spacy.load("en_core_web_sm")

STOPWORDS = set(stopwords.words('english'))


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

# Utilities: Text preprocessing

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

def clean_text(text: str) -> str:

    """Basic cleaning: remove urls, mentions, hashtags (keep text), punctuation, extra spaces."""

    text = re.sub(r"http\S+|www\S+|https\S+", "", text, flags=re.MULTILINE)

    text = re.sub(r"@\w+", "", text)  # remove mentions

    text = re.sub(r"#", "", text)     # remove hash symbol but keep the word

    text = re.sub(r"[^A-Za-z0-9(),!?\'\`\"]+", " ", text)

    text = re.sub(r"\s{2,}", " ", text)

    text = text.strip().lower()

    return text


def preprocess_text(text: str) -> str:

    text = clean_text(text)

    # lemmatize & remove stopwords

    doc = nlp(text)

    tokens = [token.lemma_ for token in doc if token.lemma_ not in STOPWORDS and token.is_alpha]

    return " ".join(tokens)
```

```
# ----------------------------
# Dataset class for HuggingFace-style fine tuning
# ----------------------------
class TweetDataset(Dataset):
    def __init__(self, texts: List[str], labels: List[int], tokenizer, max_len=MAX_LEN):
        self.texts = texts
        self.labels = labels
        self.tokenizer = tokenizer
        self.max_len = max_len

    def __len__(self):
        return len(self.texts)

    def __getitem__(self, idx):
        text = str(self.texts[idx])
        label = int(self.labels[idx])
        encoding = self.tokenizer(
            text,
            add_special_tokens=True,
            truncation=True,
            max_length=self.max_len,
            padding='max_length',
            return_attention_mask=True,
            return_tensors='pt',
        )
        return {
```

```python
            'input_ids': encoding['input_ids'].flatten(),
            'attention_mask': encoding['attention_mask'].flatten(),
            'labels': torch.tensor(label, dtype=torch.long)
        }

# ----------------------------
# Model training utilities
# ----------------------------
def train_epoch(model, data_loader, optimizer, scheduler):
    model.train()
    losses = []
    correct_predictions = 0
    for batch in data_loader:
        input_ids = batch['input_ids'].to(DEVICE)
        attention_mask = batch['attention_mask'].to(DEVICE)
        labels = batch['labels'].to(DEVICE)
        outputs = model(input_ids=input_ids, attention_mask=attention_mask, labels=labels)
        loss = outputs.loss
        logits = outputs.logits
        _, preds = torch.max(logits, dim=1)
        correct_predictions += torch.sum(preds == labels)
        losses.append(loss.item())
        loss.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
        optimizer.step()
        scheduler.step()
```

```
        optimizer.zero_grad()
    return correct_predictions.double() / (len(data_loader.dataset)), np.mean(losses)


def eval_model(model, data_loader):
    model.eval()
    losses = []
    correct_predictions = 0
    preds_all = []
    labels_all = []
    with torch.no_grad():
        for batch in data_loader:
            input_ids = batch['input_ids'].to(DEVICE)
            attention_mask = batch['attention_mask'].to(DEVICE)
            labels = batch['labels'].to(DEVICE)
            outputs = model(input_ids=input_ids, attention_mask=attention_mask,
labels=labels)
            loss = outputs.loss
            logits = outputs.logits
            _, preds = torch.max(logits, dim=1)
            correct_predictions += torch.sum(preds == labels)
            preds_all.extend(preds.cpu().numpy().tolist())
            labels_all.extend(labels.cpu().numpy().tolist())
            losses.append(loss.item())
    # compute precision/recall/f1 using sklearn
    from sklearn.metrics import precision_recall_fscore_support, accuracy_score
    precision, recall, f1, _ = precision_recall_fscore_support(labels_all, preds_all,
average='weighted', zero_division=0)
```

```
    return accuracy_score(labels_all, preds_all), precision, recall, f1, np.mean(losses)


# ----------------------------
# Inference helper
# ----------------------------
def predict_texts(model, tokenizer, texts: List[str], batch_size=BATCH_SIZE) -> List[int]:
    model.eval()
    preds = []
    dataset = TweetDataset(texts, [0]*len(texts), tokenizer)  # labels dummy
    loader = DataLoader(dataset, batch_size=batch_size)
    with torch.no_grad():
        for batch in loader:
            input_ids = batch['input_ids'].to(DEVICE)
            attention_mask = batch['attention_mask'].to(DEVICE)
            outputs = model(input_ids=input_ids, attention_mask=attention_mask)
            logits = outputs.logits
            _, batch_preds = torch.max(logits, dim=1)
            preds.extend(batch_preds.cpu().numpy().tolist())
    return preds


# ----------------------------
# Graph construction & community detection
# ----------------------------
def build_interaction_graph(interactions: pd.DataFrame, user_toxicity: Dict[str, float]) ->
nx.DiGraph:
    """
```

```
    interactions: DataFrame with columns ['source_user','target_user','interaction_count']
    user_toxicity: dict user_id -> toxicity_score (0-1)
    """
    G = nx.DiGraph()
    # add nodes with toxicity attr
    users = set(interactions['source_user']).union(set(interactions['target_user']))
    for u in users:
        G.add_node(u, toxicity=user_toxicity.get(u, 0.0))
    # add edges with normalized weights
    max_count = interactions['interaction_count'].max() if not interactions.empty else 1
    for _, row in interactions.iterrows():
        w = float(row['interaction_count']) / max_count
        G.add_edge(row['source_user'], row['target_user'], weight=w)
    return G


def detect_communities(G: nx.Graph) -> Tuple[Dict[str, int], float]:
    """
    Returns partition (user->community_id) and modularity
    """
    # community_louvain expects an undirected graph for best results; convert weights
    U = G.to_undirected()
    partition = community_louvain.best_partition(U, weight='weight')
    modularity = community_louvain.modularity(partition, U, weight='weight')
    return partition, modularity
```

```
def compute_community_toxicity(partition: Dict[str,int], user_toxicity: Dict[str,float]) ->
Dict[int, float]:
    comm_sum = defaultdict(float)
    comm_count = defaultdict(int)
    for user, comm in partition.items():
        comm_sum[comm] += user_toxicity.get(user, 0.0)
        comm_count[comm] += 1
    return {comm: (comm_sum[comm] / comm_count[comm]) for comm in comm_sum}


# ----------------------------
# I/O & helper for Gephi export
# ----------------------------
def export_graph_gexf(G: nx.Graph, path: str):
    nx.write_gexf(G, path)
    print(f"[INFO] Exported graph to {path}. Open it in Gephi for detailed visualization.")


# ----------------------------
# Synthetic data generator (for testing without Twitter access)
# ----------------------------
def generate_synthetic_dataset(n_users=200, n_tweets=1000):
    users = [f"user_{i}" for i in range(n_users)]
    texts = []
    user_ids = []
    labels = []  # 0 neutral, 1 offensive, 2 hate
    for _ in range(n_tweets):
        u = random.choice(users)
```

```
    # sample label distribution
    p = random.random()
    if p < 0.6:
        lbl = 0
        text = "I love the community and its culture"
    elif p < 0.85:
        lbl = 1
        text = "I hate XYZ sometimes but not all"
    else:
        lbl = 2
        text = "Group ABC should be removed"  # synthetic hate
    texts.append(text)
    user_ids.append(u)
    labels.append(lbl)
tweets_df = pd.DataFrame({
    'tweet_id': [f"t_{i}" for i in range(n_tweets)],
    'user_id': user_ids,
    'text': texts,
    'label': labels
})
# interactions: randomly sample mentions/retweets
rows = []
for _ in range(n_tweets * 2):
    s = random.choice(users)
    t = random.choice(users)
    if s == t: continue
```

```
      rows.append({'source_user': s, 'target_user': t, 'interaction_count':
random.randint(1,5)})
    interactions_df =
pd.DataFrame(rows).groupby(['source_user','target_user']).sum().reset_index()
    return tweets_df, interactions_df


# ----------------------------
# Main pipeline function
# ----------------------------
def run_pipeline(use_synthetic=True, twitter_json_path=None):
    """
    Main function to run end-to-end pipeline.
    - use_synthetic: if True, runs on internal synthetic data (no API keys needed)
    - twitter_json_path: optional path to CSV/JSON with collected tweets if you have them
    """
    # 1) Data load
    if use_synthetic:
      tweets_df, interactions_df = generate_synthetic_dataset(n_users=500, n_tweets=3000)
      print("[INFO] Using synthetic dataset.")
    else:
      if twitter_json_path is None:
        raise ValueError("Provide a path to your collected tweets or set use_synthetic=True")
      # load tweets CSV/JSON expected to have tweet_id, user_id, text, label (optional)
      tweets_df = pd.read_json(twitter_json_path, lines=True) if
twitter_json_path.endswith('.jsonl') else pd.read_csv(twitter_json_path)
      # You should prepare interactions_df yourself from metadata (mentions, retweets)
      interactions_df = pd.DataFrame()  # placeholder
```

```
        print("[INFO] Loaded provided twitter data.")

    # Preprocess texts
    tweets_df['clean_text'] = tweets_df['text'].astype(str).apply(preprocess_text)

    # If labels missing and synthetic False, you must annotate or use weak labeling.
    if 'label' not in tweets_df.columns:
        raise ValueError("No labels in dataset. Provide labeled data or use synthetic mode.")

    # Split into train/val
    from sklearn.model_selection import train_test_split
    train_df, val_df = train_test_split(tweets_df, test_size=0.2, stratify=tweets_df['label'],
random_state=RANDOM_SEED)

    # Tokenizer & model
    tokenizer = BertTokenizerFast.from_pretrained(MODEL_NAME)
    model = BertForSequenceClassification.from_pretrained(MODEL_NAME,
num_labels=3).to(DEVICE)

    # Datasets & loaders
    train_dataset = TweetDataset(train_df['clean_text'].tolist(), train_df['label'].tolist(),
tokenizer)
    val_dataset = TweetDataset(val_df['clean_text'].tolist(), val_df['label'].tolist(), tokenizer)
    train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True)
    val_loader = DataLoader(val_dataset, batch_size=BATCH_SIZE)

    # Optimizer & scheduler
```

```
    optimizer = AdamW(model.parameters(), lr=LR)
    total_steps = len(train_loader) * EPOCHS
    scheduler = get_linear_schedule_with_warmup(optimizer,
num_warmup_steps=int(0.1*total_steps), num_training_steps=total_steps)

    # Train loop (small number of epochs for demo; increase for production)
    best_f1 = 0.0
    for epoch in range(EPOCHS):
        train_acc, train_loss = train_epoch(model, train_loader, optimizer, scheduler)
        val_acc, val_prec, val_rec, val_f1, val_loss = eval_model(model, val_loader)
        print(f"Epoch {epoch+1}/{EPOCHS} — train_acc: {train_acc:.4f}, train_loss:
{train_loss:.4f} | val_acc: {val_acc:.4f}, val_f1: {val_f1:.4f}")
        if val_f1 > best_f1:
            best_f1 = val_f1
            # save best model
            model_save_path = "best_hate_model.pt"
            torch.save(model.state_dict(), model_save_path)
            print(f"[INFO] Saved best model to {model_save_path}")

    # Load best model for inference
    model.load_state_dict(torch.load("best_hate_model.pt"))
    model.to(DEVICE)

    # 2) Inference on all tweets
    all_texts = tweets_df['clean_text'].tolist()
    preds = predict_texts(model, tokenizer, all_texts, batch_size=BATCH_SIZE)
    tweets_df['pred_label'] = preds
```

```
    # Map labels to toxicity: neutral=0, offensive=0.5, hate=1.0
    label_to_toxicity = {0: 0.0, 1: 0.5, 2: 1.0}
    user_group = tweets_df.groupby('user_id').agg({'pred_label': list, 'tweet_id':
'count'}).reset_index()
    def user_toxicity_from_preds(preds_list):
        return np.mean([label_to_toxicity[p] for p in preds_list]) if preds_list else 0.0
    user_group['toxicity_score'] = user_group['pred_label'].apply(user_toxicity_from_preds)


    user_toxicity = dict(zip(user_group['user_id'], user_group['toxicity_score']))


    # 3) Build or normalize interactions DataFrame
    # If synthetic, interactions_df already present. Ensure columns
source_user,target_user,interaction_count
    if interactions_df.empty:
        # create a simple interactions df from tweets: random mention graph for synthetic
demo
        interactions = []
        users = list(user_group['user_id'])
        for u in users:
            for _ in range(random.randint(1,4)):
                v = random.choice(users)
                if v != u:
                    interactions.append({'source_user': u, 'target_user': v, 'interaction_count':
random.randint(1,4)})
        interactions_df =
pd.DataFrame(interactions).groupby(['source_user','target_user']).sum().reset_index()
```

```
    # 4) Graph construction
    G = build_interaction_graph(interactions_df, user_toxicity)
    print(f"[INFO] Graph constructed: {G.number_of_nodes()} nodes, {G.number_of_edges()}
edges")

    # 5) Community detection
    partition, modularity = detect_communities(G)
    print(f"[INFO] Detected {len(set(partition.values()))} communities with
modularity={modularity:.4f}")

    # compute community toxicity
    comm_toxicity = compute_community_toxicity(partition, user_toxicity)
    # label nodes with community and community_toxicity
    for node in G.nodes():
        comm_id = partition.get(node, -1)
        G.nodes[node]['community'] = int(comm_id)
        G.nodes[node]['community_toxicity'] = float(comm_toxicity.get(comm_id, 0.0))

    # identify high-toxicity communities
    high_risk = {cid: t for cid, t in comm_toxicity.items() if t > 0.7}
    print(f"[INFO] High-risk communities (toxicity>0.7): {len(high_risk)}")

    # 6) Export for Gephi and quick plot
    export_graph_gexf(G, "social_graph.gexf")

    # quick matplot visualization colored by community toxicity (coarse)
```

```python
plt.figure(figsize=(10,8))
und = G.to_undirected()
pos = nx.spring_layout(und, seed=RANDOM_SEED)
node_colors = []
for n in und.nodes():
   t = und.nodes[n].get('community_toxicity', 0.0)
   # color mapping: red high, yellow mid, green low
   if t > 0.7:
      node_colors.append('red')
   elif t > 0.4:
      node_colors.append('orange')
   else:
      node_colors.append('green')
nx.draw_networkx_nodes(und, pos, node_size=30, node_color=node_colors, alpha=0.8)
nx.draw_networkx_edges(und, pos, alpha=0.2)
plt.title("Social Graph (node color by community toxicity)")
plt.axis('off')
plt.show()

# Save summary CSVs
user_summary = pd.DataFrame({
   'user_id': list(user_toxicity.keys()),
   'toxicity_score': list(user_toxicity.values()),
   'community': [partition.get(u, -1) for u in user_toxicity.keys()]
})
user_summary.to_csv("user_summary.csv", index=False)
```

```python
    print("[INFO] user_summary.csv saved. Pipeline complete.")


if __name__ == "__main__":
    # Set use_synthetic=False and provide real data path if you have collected tweets
    run_pipeline(use_synthetic=True)
```