Shahd Mohamed Abd El Sabour

2205027

Assignment 2 Social Networking

Overview :

This assignment builds a social graph (tries to download the SNAP Facebook network, otherwise creates a synthetic SBM graph), computes per-node graph features, synthetically labels a small fraction of nodes as “bots”, trains a Random Forest classifier to detect bots, then simulates two adversarial attacks:

- **Structural evasion**: bots change their connectivity (remove some edges to high-degree neighbors and add edges to low-degree nodes) to try to evade detection.
- **Graph poisoning**: the training data is corrupted by flipping labels and injecting fake nodes with many edges; the model is retrained on this poisoned data and evaluated on the clean test set.

Assignment2 Social.py × graph_original.png

C: > Users > shahd > OneDrive > Desktop > Assignment2 Social.py > graph_poisoning

```python
import os
import gzip
import random
import urllib.request
import math
from collections import defaultdict
import argparse
import joblib

import networkx as nx
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.metrics import classification_report, accuracy_score, roc_auc_score

```

- Standard libs for files, randomness, downloading.
- `networkx` for graph operations.
- `pandas`/`numpy` for data handling.
- `sklearn` for modeling and metrics.
- `joblib` to save models.

```python
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.metrics import classification_report, accuracy_score, roc_auc_score

# ------------------------
# Configurable parameters
# ------------------------
SNAP_URL = "https://snap.stanford.edu/data/facebook_combined.txt.gz"
OUTPUT_DIR = "outputs"
RANDOM_SEED = 42
```

`RANDOM_SEED` used to make behavior reproducible (where supported). The script also sets Python and numpy seeds.

```python

# ------------------------
# Utilities
# ------------------------
def ensure_outdir(path=OUTPUT_DIR):
    os.makedirs(path, exist_ok=True)

def try_download_snap(save_to):
    try:
        print("[*] Attempting to download SNAP facebook_combined dataset...")
        urllib.request.urlretrieve(SNAP_URL, save_to)
        print("[+] Downloaded SNAP dataset to:", save_to)
        return True
    except Exception as e:
        print("[!] Could not download SNAP dataset (internet blocked or error):", e)
        return False

def load_snap_from_gz(path):
    edges = []
    with gzip.open(path, "rt") as f:
        for line in f:
            if not line.strip():
                continue
            a,b = line.split()
            edges.append((int(a), int(b)))
    return edges
```

`ensure_outdir(path=OUTPUT_DIR)`

Creates the output directory if it doesn't exist.

`try_download_snap(save_to)`

Attempts to download the SNAP facebook dataset to `save_to`. Returns True on success, False on any exception (e.g., no internet).

`load_snap_from_gz(path)`

Reads the gzipped edge list line by line and returns a list of `(a,b)` integer edges. Used if the SNAP download succeeds.

```
            a,b = line.split()
            edges.append((int(a), int(b)))
    return edges

def build_sbm_graph(sizes=None, p_in=0.1, p_out=0.005):
    """Build a synthetic social graph using stochastic block model (SBM)."""
    if sizes is None:
        sizes = [400,350,300,250,200,200,300]  # ~2000 nodes
    # Build probability matrix
    nblocks = len(sizes)
    p = [[p_in if i==j else p_out for j in range(nblocks)] for i in range(nblocks)]
    G = nx.stochastic_block_model(sizes, p, seed=RANDOM_SEED)
    G = nx.convert_node_labels_to_integers(G)
    return G

```

`build_sbm_graph(sizes=None, p_in=0.1, p_out=0.005)`

- Builds a Stochastic Block Model (SBM) with `sizes` blocks (default block sizes sum to ~2000 nodes).
- Constructs a probability matrix p where intra-block edge prob = `p_in` and inter-block prob = `p_out`.
- Calls `nx.stochastic_block_model` and re-labels nodes to consecutive integers.
- Returns a `networkx.Graph`.

Use-case: fallback if SNAP download or load fails.

Assignment2 Social.py × graph_original.png

C: > Users > shahd > OneDrive > Desktop > Assignment2 Social.py > compute_graph_metrics

```python

# ------------------------
# Feature extraction
# ------------------------
def compute_graph_metrics(G, approx_betweenness_k=200):
    nodes = list(G.nodes())
    print("[*] Computing degree and clustering...")
    deg = dict(G.degree(nodes))
    clustering = nx.clustering(G)  # returns dict

    # Betweenness: approximate when graph is large
    n = G.number_of_nodes()
    betweenness = {}
    try:
        if n > 1000:
            # use k-approximation sampling for speed
            k = min(approx_betweenness_k, n)
            print(f"[*] Approximating betweenness centrality with k={k} samples...")
            betweenness = nx.betweenness_centrality(G, k=k, seed=RANDOM_SEED)
        else:
            print("[*] Computing exact betweenness centrality...")
            betweenness = nx.betweenness_centrality(G)
    except Exception as e:
        print("[!] Betweenness computation failed, setting zeros. Error:", e)
        betweenness = {node: 0.0 for node in nodes}

    # Eigenvector centrality using numpy
    try:
        print("[*] Computing eigenvector centrality (numpy)...")
        eigen = nx.eigenvector_centrality_numpy(G)
```

Ln 73, Col 26 Spaces: 4 UTF-8 CRLF { } Python Signed out 3.13.9 (Microsoft Store)

Assignment2 Social.py × graph_original.png

C: > Users > shahd > OneDrive > Desktop > Assignment2 Social.py > compute_graph_metrics

```python
def compute_graph_metrics(G, approx_betweenness_k=200):
        print("[*] Computing eigenvector centrality (numpy)...")
        eigen = nx.eigenvector_centrality_numpy(G)
    except Exception as e:
        print("[!] Eigenvector failed, falling back to degree-normalized proxy. Error:", e)
        eigen = {node: float(deg.get(node,0)) / (max(deg.values())+1e-6) for node in nodes}

    print("[*] Computing average neighbor degree and core numbers...")
    avg_neighbor_degree = nx.average_neighbor_degree(G)
    try:
        core_number = nx.core_number(G)
    except Exception as e:
        print("[!] core_number failed, filling zeros. Error:", e)
        core_number = {node: 0 for node in nodes}

    # Build DataFrame
    data = {
        'node': nodes,
        'degree': [deg.get(n, 0) for n in nodes],
        'clustering': [clustering.get(n, 0.0) for n in nodes],
        'betweenness': [betweenness.get(n, 0.0) for n in nodes],
        'eigenvector': [eigen.get(n, 0.0) for n in nodes],
        'avg_neighbor_degree': [avg_neighbor_degree.get(n, 0.0) for n in nodes],
        'k_core': [core_number.get(n, 0) for n in nodes],
    }
    df = pd.DataFrame(data).set_index('node')
    return df

# ------------------------
# Bot labeling (synthetic)
```

Ln 119, Col 6 Spaces: 4 UTF-8 CRLF { } Python Signed out 3.13.9 (Microsoft Store)

`compute_graph_metrics(G, approx_betweenness_k=200)`

For graph G, computes per-node features and returns a `pandas.DataFrame` indexed by node id. Features:

- `degree`: node degree (number of neighbors).
- `clustering`: clustering coefficient (triangles / possible triangles).
- `betweenness`: betweenness centrality. If `n > 1000`, uses k-sample approximate betweenness via `nx.betweenness_centrality(G, k=k, seed=RANDOM_SEED)` for speed; otherwise computes exact.
    - The code catches exceptions and falls back to zeros on failure.
- `eigenvector`: eigenvector centrality using `nx.eigenvector_centrality_numpy`. On failure falls back to a proxy: degree normalized by max degree.
- `avg_neighbor_degree`: average neighbor degree (from NetworkX).
- `k_core`: core number (`nx.core_number`). On failure fills zeros.

Important performance notes:

- Exact betweenness and eigenvector centralities can be expensive on large graphs; the script attempts approximate betweenness when `n > 1000`.

Return: DataFrame with one row per node and columns listed above.

```python
    return df

# ------------------------
# Bot labeling (synthetic)
# ------------------------
def label_synthetic_bots(df, fraction=0.05):
    nodes = list(df.index)
    n = len(nodes)
    num_bots = max(1, int(fraction * n))
    # pick half top-degree, half random
    sorted_by_deg = df.sort_values('degree', ascending=False).index.tolist()
    half = num_bots // 2
    high_deg_bots = sorted_by_deg[:half]
    remaining = list(set(nodes) - set(high_deg_bots))
    random_bots = random.sample(remaining, num_bots - len(high_deg_bots))
    bot_nodes = set(high_deg_bots + random_bots)
    labels = pd.Series(0, index=df.index)
    labels.loc[list(bot_nodes)] = 1
    return bot_nodes, labels
```

`label_synthetic_bots(df, fraction=0.05)`

- `fraction` fraction of nodes are labeled as bots (default 5%).
- Strategy: choose half of bot nodes as the highest-degree nodes, and the other half randomly from the rest.
- Returns `bot_nodes` (a `set`) and `labels` (a `pandas.Series` indexed by nodes with 1 for bot, 0 otherwise).

This creates a plausible synthetic scenario where some bots are high-degree (influencer-like) and others look random.

```python
    return bot_nodes, labels

# ------------------------
# Model training & eval
# ------------------------
def train_evaluate(X_train, y_train, X_test, y_test, return_model=True):
    clf = Pipeline([
        ('scaler', StandardScaler()),
        ('rf', RandomForestClassifier(n_estimators=200, random_state=RANDOM_SEED))
    ])
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)
    y_proba = clf.predict_proba(X_test)[:,1] if hasattr(clf, "predict_proba") else None
    report = classification_report(y_test, y_pred, output_dict=True)
    acc = accuracy_score(y_test, y_pred)
    auc = roc_auc_score(y_test, y_proba) if y_proba is not None and len(set(y_test))>1 else float('na
    return {
        'model': clf if return_model else None,
        'report': report,
        'accuracy': acc,
        'auc': auc,
        'y_pred': y_pred,
        'y_proba': y_proba
    }
```

`train_evaluate(X_train, y_train, X_test, y_test, return_model=True)`

- Builds a `Pipeline` with `StandardScaler` + `RandomForestClassifier(n_estimators=200, random_state=RANDOM_SEED)`.
- Trains on training data, predicts on test data.
- Computes `classification_report` (as dict), accuracy, and AUC (if possible).
- Returns dictionary containing model, report, accuracy, auc, `y_pred`, `y_proba`.

Note: if all test labels are the same, ROC AUC is not computable and the code uses nan.

C: > Users > shahd > OneDrive > Desktop > Assignment2 Social.py > structural_evasion

```python
    }

# ------------------------
# Attacks
# ------------------------
def structural_evasion(G, bot_nodes, remove_frac=0.5, add_degree=5):
    G2 = G.copy()
    for b in bot_nodes:
        if b not in G2:
            continue
        nbrs = list(G2.neighbors(b))
        if not nbrs:
            continue
        # remove top-degree neighbors
        nbrs_sorted = sorted(nbrs, key=lambda x: G2.degree[x], reverse=True)
        num_remove = max(1, int(remove_frac * len(nbrs_sorted)))
        to_remove = nbrs_sorted[:num_remove]
        for r in to_remove:
            if G2.has_edge(b, r):
                G2.remove_edge(b, r)
        # add edges to low-degree nodes
        candidates = [n for n in G2.nodes() if n != b and not G2.has_edge(b, n)]
        candidates_sorted = sorted(candidates, key=lambda x: G2.degree[x])
        to_add = candidates_sorted[:add_degree] if len(candidates_sorted) >= add_degree else random.s
        for a in to_add:
            G2.add_edge(b, a)
    return G2

```

C: > Users > shahd > OneDrive > Desktop > Assignment2 Social.py > graph_poisoning

```python
def structural_evasion(G, bot_nodes, remove_frac=0.5, add_degree=5):
    return G2

def graph_poisoning(G, train_nodes, flip_frac=0.15, inject_frac=0.02, edges_per_inject=20):
    G2 = G.copy()
    train_nodes = list(train_nodes)
    num_flip = max(1, int(flip_frac * len(train_nodes)))
    flipped = random.sample(train_nodes, num_flip)

    # Inject nodes
    n_inject = max(1, int(inject_frac * G.number_of_nodes()))
    start_idx = max(G2.nodes()) + 1 if len(G2.nodes())>0 else 0
    injected = list(range(start_idx, start_idx + n_inject))
    for node in injected:
        G2.add_node(node)
    # Connect injected nodes to many high-degree targets
    high_deg = sorted(G2.nodes(), key=lambda x: G2.degree[x], reverse=True)[:200]
    if not high_deg:
        high_deg = list(G2.nodes())
    for inj in injected:
        targets = random.sample(high_deg, min(edges_per_inject, len(high_deg)))
        for t in targets:
            G2.add_edge(inj, t)
    return G2, flipped, injected

```

`structural_evasion(G, bot_nodes, remove_frac=0.5, add_degree=5)`

- For each bot b:
  - Get its neighbors and sort them by neighbor degree descending.
  - Remove the top `remove_frac` fraction of these neighbors (i.e., disconnect from high-degree neighbors).
  - Add `add_degree` new edges from b to low-degree nodes it wasn't connected to (candidates sorted by degree ascending). If there are not enough low-degree nodes, it picks a random subset.
- Returns a new graph G2 with the structural changes.
- Goal: make bot nodes less obviously high-degree/hub-linked and more blended with low-degree nodes.

`graph_poisoning(G, train_nodes, flip_frac=0.15, inject_frac=0.02, edges_per_inject=20)`

- Copies the graph to G2.
- Chooses `num_flip = flip_frac * len(train_nodes)` training nodes and marks them as `flipped` (this list is returned but not immediately applied to labels here — the caller flips labels in the training set).
- Injects `n_inject = inject_frac * number_of_nodes` new nodes with consecutive integer ids beyond current max node id.
- Connects each injected node to many high-degree targets (top 200 high-degree nodes) with up to `edges_per_inject` edges each.
- Returns `(G2, flipped, injected)`.

Purpose: emulate poisoning the training graph by injecting noisy nodes and prepare to flip labels for some training nodes.

C: > Users > shahd > OneDrive > Desktop > Assignment2 Social.py > plot_graph_sample

```python

# ------------------------
# Visualization
# ------------------------
def plot_graph_sample(G, bot_nodes=None, outpath=None, sample_size=500, title="Graph sample"):
    if G.number_of_nodes() == 0:
        return
    if G.number_of_nodes() > sample_size:
        sample = random.sample(list(G.nodes()), sample_size)
        H = G.subgraph(sample).copy()
    else:
        H = G.copy()
    pos = nx.spring_layout(H, seed=RANDOM_SEED)
    plt.figure(figsize=(8,8))
    if bot_nodes is None:
        nx.draw_networkx_nodes(H, pos, node_size=20)
    else:
        node_color = []
        for n in H.nodes():
            node_color.append(1 if n in bot_nodes else 0)
        nx.draw_networkx_nodes(H, pos, node_size=20, cmap=plt.cm.viridis, node_color=node_color)
    nx.draw_networkx_edges(H, pos, alpha=0.3)
    plt.title(title)
    plt.axis('off')
    if outpath:
        plt.savefig(outpath, bbox_inches='tight', dpi=200)
        plt.close()
    else:
        plt.show()

```

`plot_graph_sample(G, bot_nodes=None, outpath=None, sample_size=500, title="Graph sample")`

- If the graph has more nodes than `sample_size`, sample `sample_size` node IDs and work with the induced subgraph to keep drawing fast.
- Uses `nx.spring_layout` (seeded) to get positions and draws nodes and edges.
- When `bot_nodes` is given, nodes are colored by whether they're in the `bot_nodes` set (passed values 0/1 to `node_color`).
- Saves to `outpath` if provided, otherwise shows plot.

C: > Users > shahd > OneDrive > Desktop > Assignment2 Social.py > main_flow

```python
# Main pipeline
# ------------------------
def main_flow(force_synthetic=False):
    ensure_outdir(OUTPUT_DIR)
    snap_gz = os.path.join(OUTPUT_DIR, "facebook_combined.txt.gz")
    G = None

    if not force_synthetic:
        downloaded = try_download_snap(snap_gz)
        if downloaded:
            try:
                edges = load_snap_from_gz(snap_gz)
                print("[*] Building graph from SNAP edges...")
                G = nx.Graph()
                G.add_edges_from(edges)
            except Exception as e:
                print("[!] Failed loading SNAP edges, will build synthetic graph. Error:", e)
                G = None

    if G is None:
        print("[*] Building synthetic SBM graph (fallback)...")
        G = build_sbm_graph()

    print(f"[+] Graph: nodes={G.number_of_nodes()}, edges={G.number_of_edges()}")

    # Compute baseline features
    df_features = compute_graph_metrics(G)
    df_features.to_csv(os.path.join(OUTPUT_DIR, "node_features_original.csv"))

    # Label bots (synthetic)
```

Assignment2 Social.py × graph_original.png

C: > Users > shahd > OneDrive > Desktop > Assignment2 Social.py > main_flow

```python
def main_flow(force_synthetic=False):

    # Label bots (synthetic)
    bot_nodes, labels = label_synthetic_bots(df_features, fraction=0.05)
    df_features['is_bot'] = labels

    # Split train/test (stratify)
    FEATURE_COLS = ['degree','clustering','betweenness','eigenvector','avg_neighbor_degree','k_core']
    X = df_features[FEATURE_COLS].fillna(0).values
    y = df_features['is_bot'].values
    node_ids = df_features.index.values

    X_train, X_test, y_train, y_test, node_train, node_test = train_test_split(
        X, y, node_ids, test_size=0.3, random_state=RANDOM_SEED, stratify=y
    )

    # Baseline model
    print("[*] Training baseline classifier...")
    baseline = train_evaluate(X_train, y_train, X_test, y_test, return_model=True)
    joblib.dump(baseline['model'], os.path.join(OUTPUT_DIR, "baseline_model.joblib"))

    # Save baseline report
    with open(os.path.join(OUTPUT_DIR, "baseline_results.txt"), "w") as f:
        f.write(classification_report(y_test, baseline['y_pred']))
        f.write(f"\nAccuracy: {baseline['accuracy']}\nAUC: {baseline['auc']}\n")

    print("[+] Baseline accuracy:", baseline['accuracy'], "AUC:", baseline['auc'])

    # Visualize original graph (sample)
    plot_graph_sample(G, bot_nodes=bot_nodes, outpath=os.path.join(OUTPUT_DIR, "graph_original.png"),
```

Ln 293, Col 26 Spaces: 4 UTF-8 CRLF { } Python Signed out 3.13.9 (Microsoft Store)

C: > Users > shahd > OneDrive > Desktop > Assignment2 Social.py > main_flow

```python
def main_flow(force_synthetic=False):
    joblib.dump(baseline['model'], os.path.join(OUTPUT_DIR, "baseline_model.joblib"))

    # Save baseline report
    with open(os.path.join(OUTPUT_DIR, "baseline_results.txt"), "w") as f:
        f.write(classification_report(y_test, baseline['y_pred']))
        f.write(f"\nAccuracy: {baseline['accuracy']}\nAUC: {baseline['auc']}\n")

    print("[+] Baseline accuracy:", baseline['accuracy'], "AUC:", baseline['auc'])

    # Visualize original graph (sample)
    plot_graph_sample(G, bot_nodes=bot_nodes, outpath=os.path.join(OUTPUT_DIR, "graph_original.png"),

    # -------------------------
    # Structural evasion
    # -------------------------
    attacked_subset = random.sample(list(bot_nodes), max(1, int(0.3*len(bot_nodes))))  # attack 30% c
    G_struct = structural_evasion(G, attacked_subset, remove_frac=0.5, add_degree=5)
    df_struct = compute_graph_metrics(G_struct)
    # Ensure ordering same as original nodes for comparison
    df_struct = df_struct.reindex(df_features.index).fillna(0)
    df_struct.to_csv(os.path.join(OUTPUT_DIR, "node_features_structural.csv"))

    # Evaluate baseline model on features computed after structural evasion (no retraining)
    X_struct_test = df_struct.loc[node_test, FEATURE_COLS].fillna(0).values
    struct_eval = {}
    # reuse baseline model to predict
    model_baseline = baseline['model']
    y_struct_pred = model_baseline.predict(X_struct_test)
    try:
```

Ln 311, Col 26   Spaces: 4   UTF-8   CRLF   { } Python   Signed out   3.13.9 (Microsoft Store)

C: > Users > shahd > OneDrive > Desktop > Assignment2 Social.py > main_flow > flipped_nodes

```python
def main_flow(force_synthetic=False):
    struct_eval = {}
    # reuse baseline model to predict
    model_baseline = baseline['model']
    y_struct_pred = model_baseline.predict(X_struct_test)
    try:
        y_struct_proba = model_baseline.predict_proba(X_struct_test)[:,1]
    except Exception:
        y_struct_proba = None
    struct_report = classification_report(y_test, y_struct_pred, output_dict=False)
    struct_acc = accuracy_score(y_test, y_struct_pred)
    struct_auc = roc_auc_score(y_test, y_struct_proba) if y_struct_proba is not None and len(set(y_te

    with open(os.path.join(OUTPUT_DIR, "structural_attack_results.txt"), "w") as f:
        f.write(struct_report)
        f.write(f"\nAccuracy: {struct_acc}\nAUC: {struct_auc}\n")

    print("[+] After structural evasion -> accuracy:", struct_acc, "AUC:", struct_auc)
    plot_graph_sample(G_struct, bot_nodes=bot_nodes, outpath=os.path.join(OUTPUT_DIR, "graph_structur

    # -------------------------
    # Graph poisoning
    # -------------------------
    # We'll poison training set: flip some labels and inject nodes, then retrain
    G_poisoned, flipped_nodes, injected_nodes = graph_poisoning(G, node_train, flip_frac=0.15, inject
    df_poison = compute_graph_metrics(G_poisoned)
    df_poison = df_poison.reindex(df_features.index).fillna(0)
    df_poison.to_csv(os.path.join(OUTPUT_DIR, "node_features_poison.csv"))

    # Prepare poisoned training labels: flip labels for 'flipped_nodes' that are in train
```

Assignment2 Social.py × graph_original.png

C: > Users > shahd > OneDrive > Desktop > Assignment2 Social.py > main_flow

```python
def main_flow(force_synthetic=False):

    # Prepare poisoned training labels: flip labels for 'flipped_nodes' that are in train
    # Find positions within node_train
    y_train_poison = y_train.copy()
    node_train_list = list(node_train)
    for n in flipped_nodes:
        if n in node_train_list:
            idx = node_train_list.index(n)
            y_train_poison[idx] = 1 - y_train_poison[idx]  # flip

    # Build poisoned training X using new features computed on poisoned graph
    train_positions = [list(df_poison.index).index(n) for n in node_train]  # mapping node id -> row
    X_train_poison = df_poison.iloc[train_positions][FEATURE_COLS].fillna(0).values

    # Retrain classifier on poisoned data
    clf_poisoned = Pipeline([
        ('scaler', StandardScaler()),
        ('rf', RandomForestClassifier(n_estimators=200, random_state=RANDOM_SEED))
    ])
    print("[*] Training classifier on poisoned training data...")
    clf_poisoned.fit(X_train_poison, y_train_poison)

    # Evaluate poisoned-trained model on clean test features (from original df_features)
    X_test_clean = df_features.loc[node_test, FEATURE_COLS].fillna(0).values
    y_poisoned_pred = clf_poisoned.predict(X_test_clean)
    try:
        y_poisoned_proba = clf_poisoned.predict_proba(X_test_clean)[:,1]
    except Exception:
        y_poisoned_proba = None
```

1. **Prepare outputs**: call `ensure_outdir(OUTPUT_DIR)`.
2. **Download / load graph**:
    a. If `force_synthetic` is `False`, try to download the SNAP file to `outputs/facebook_combined.txt.gz`.
    b. If download succeeds, load edges using `load_snap_from_gz` and create `G = nx.Graph()` with `G.add_edges_from(edges)`.
    c. If any step fails, or if `force_synthetic=True`, fall back to `G = build_sbm_graph()` (the synthetic SBM).
3. **Print graph size**: nodes and edges.
4. **Compute baseline features**:
    a. `df_features = compute_graph_metrics(G)` and save to `node_features_original.csv`.
5. **Label bots**:
    a. `bot_nodes, labels = label_synthetic_bots(df_features, fraction=0.05)`.
    b. Add labels as column `is_bot` in `df_features`.
6. **Prepare feature matrix and split**:

a. FEATURE_COLS list: ['degree','clustering','betweenness','eigenvector','avg_neighbor_degree','k_core'].
b. X = feature values, y = is_bot, node_ids = index (node ids).
c. train_test_split(..., test_size=0.3, random_state=RANDOM_SEED, stratify=y) returns X_train, X_test, y_train, y_test, node_train, node_test.
    i. NOTE: stratify ensures similar bot fraction in both sets.

7. **Train baseline model**:
    a. Calls train_evaluate(...) and saves model as baseline_model.joblib.
    b. Writes a text file baseline_results.txt containing classification report, accuracy, and AUC.
    c. Prints baseline accuracy and AUC.
8. **Visualize original graph**:
    a. Calls plot_graph_sample(..., outpath="outputs/graph_original.png").
9. **Structural evasion experiment**:
    a. Select attacked_subset — 30% of bot_nodes at random to attack.
    b. G_struct = structural_evasion(G, attacked_subset, remove_frac=0.5, add_degree=5).
    c. Compute features on G_struct → df_struct and reindex to original index.
    d. Save node_features_structural.csv.
    e. Evaluate the **baseline** model on the *evaded* features for the test nodes (no retraining).
        i. Build X_struct_test from df_struct.loc[node_test, FEATURE_COLS].
        ii. y_struct_pred = model_baseline.predict(X_struct_test) and compute metrics.
    f. Save structural_attack_results.txt and print the results.
    g. Plot sample graph_structural_attack.png.
10. **Graph poisoning experiment**:
    a. G_poisoned, flipped_nodes, injected_nodes = graph_poisoning(G, node_train, flip_frac=0.15, inject_frac=0.02, edges_per_inject=20).
    b. Compute features on G_poisoned → df_poison and reindex.
    c. Save node_features_poison.csv.
    d. **Flip labels** in the training set:

        i. Copy y_train to y_train_poison.
        ii. For every n in `flipped_nodes` that appears in node_train, flip the label at that index (`1 -> 0`, `0 -> 1`).
    e. Rebuild poisoned training X_train_poison using the new features `df_poison` for nodes in node_train.
        i. The code maps nodes in node_train to their row positions in `df_poison` using `train_positions = [list(df_poison.index).index(n) for n in node_train]`.
    f. Retrain a fresh RandomForest pipeline (same config) on `(X_train_poison, y_train_poison)`.
    g. Evaluate this poisoned-trained model on the **clean** test features `X_test_clean = df_features.loc[node_test, FEATURE_COLS]`.
    h. Save `poisoning_attack_results.txt`, dump `poisoned_model.joblib`, and plot `graph_poisoning.png`.
    i. Print results.
11. **Save a Markdown report**:
    a. `report_summary.md` contains counts and summarized metrics for baseline, structural evasion, and poisoning experiments and lists produced files.
12. **Finish**: prints that outputs are saved.

```
# -------------------------
# CLI
# -------------------------
def parse_args():
    p = argparse.ArgumentParser(description="Assignment 2: Bot detection + attacks pipeline")
    p.add_argument("--force-synthetic", action="store_true", help="Skip SNAP download and force synth
    return p.parse_args()

if __name__ == "__main__":
    args = parse_args()
    main_flow(force_synthetic=args.force_synthetic)
```

`parse_args()` uses argparse to accept `--force-synthetic`.

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS CODE REFERENCE LOG Python Debug Console

```
rs\shahd\.vscode\extensions\ms-python.debugpy-2025.16.0-win32-x64\bundled\libs\debugpy\launcher' '59837' '--' 'c:\Users\sh
ahd\OneDrive\Desktop\Assignment2 Social.py'
PS C:\Users\shahd\OneDrive\Desktop\outputs>  & 'c:\Users\shahd\AppData\Local\Microsoft\WindowsApps\python3.13.exe' 'c:\Use
rs\shahd\.vscode\extensions\ms-python.debugpy-2025.16.0-win32-x64\bundled\libs\debugpy\launcher' '59837' '--' 'c:\Users\sh
ahd\OneDrive\Desktop\Assignment2 Social.py'
[*] Attempting to download SNAP facebook_combined dataset...
[+] Downloaded SNAP dataset to: outputs\facebook_combined.txt.gz
[*] Building graph from SNAP edges...
[+] Graph: nodes=4039, edges=88234
[*] Computing degree and clustering...
[*] Approximating betweenness centrality with k=200 samples...
[*] Computing eigenvector centrality (numpy)...
[*] Computing average neighbor degree and core numbers...
[*] Training baseline classifier...
[+] Baseline accuracy: 0.971947194719472 AUC: 0.7526475694444444
[*] Computing degree and clustering...
[*] Approximating betweenness centrality with k=200 samples...
[*] Computing eigenvector centrality (numpy)...
[*] Computing average neighbor degree and core numbers...
[+] After structural evasion -> accuracy: 0.9570957095709571 AUC: 0.7396122685185185
[*] Computing degree and clustering...
[*] Approximating betweenness centrality with k=200 samples...
[*] Computing eigenvector centrality (numpy)...
[*] Computing average neighbor degree and core numbers...
[*] Training classifier on poisoned training data...
[+] Poisoned training -> accuracy on clean test: 0.9645214521452146 AUC: 0.7565972222222221
[+] All outputs saved to: outputs
PS C:\Users\shahd\OneDrive\Desktop\outputs>
```

Ln 413, Col 1 Spaces: 4 UTF-8 CRLF { } Python Signed out 3.13.9 (Microsoft Store)

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	node	degree	clustering	betweenn	eigenvecto	avg_neighl	k_core								
2	0	347	0.041962	0.147377	3.31E-05	18.95965	21								
3	1	17	0.419118	1.56E-06	5.97E-07	48.23529	13								
4	2	10	0.888889	1.55E-07	2.17E-07	49.9	9								
5	3	17	0.632353	2.41E-06	6.56E-07	59.76471	13								
6	4	10	0.866667	0	2.17E-07	42.6	9								
7	5	13	0.333333	7.10E-06	1.18E-06	50.61538	10								
8	6	6	0.933333	0	2.11E-07	63.5	5								
9	7	20	0.431579	8.23E-05	2.57E-05	45.9	12								
10	8	8	0.678571	0	2.13E-07	48.375	5								
11	9	57	0.397243	6.81E-06	2.20E-06	42.40351	21								
12	10	10	0.822222	0	7.64E-07	79.1	10								
13	11	1	0	0	2.04E-07	347	1								
14	12	1	0	0	2.04E-07	347	1								
15	13	31	0.651613	8.21E-07	1.09E-06	54.54839	21								
16	14	15	0.742857	3.83E-07	2.24E-07	38.66667	10								
17	15	1	0	0	2.04E-07	347	1								
18	16	9	0.666667	0	2.55E-07	66.88889	9								
19	17	13	0.730769	3.54E-07	2.21E-07	42.76923	9								
20	18	1	0	0	2.04E-07	347	1								
21	19	16	0.283333	0	2.24E-07	33.125	7								
22	20	15	0.685714	3.61E-07	2.23E-07	37.4	9								
23	21	65	0.349038	0.001332	2.59E-05	42.4	21								
24	22	11	0.472727	0	1.03E-06	53.72727	9								

node_features_original

A1 node

	A	B	C	D	E	F	G
1	node	degree	clustering	betweenn	eigenvectc	avg_neighl	k_core
2	0	178	0.010284	0.081509	9.57E-05	5.882022	9
3	1	16	0.341667	0.000486	7.92E-07	28.5625	12
4	2	10	0.711111	0.002551	6.89E-07	32.2	9
5	3	16	0.583333	3.04E-05	2.40E-06	41	12
6	4	10	0.866667	0	7.15E-07	25.7	9
7	5	12	0.212121	0.000838	3.69E-06	25.08333	9
8	6	6	0.933333	0	6.93E-07	35.33333	5
9	7	19	0.368421	0.000174	4.52E-05	29.21053	11
10	8	8	0.678571	0	7.02E-07	27.25	5
11	9	56	0.375325	0.000245	4.01E-06	35.875	20
12	10	10	0.622222	9.89E-05	1.79E-06	61.2	9
13	11	4	0	0.000436	0.000528	94	4
14	12	4	0	0.000436	0.000528	94	4
15	13	30	0.627586	5.83E-05	1.80E-06	43.66667	20
16	14	14	0.703297	6.09E-07	2.29E-08	15.71429	9
17	15	4	0	0.000436	0.000528	94	4
18	16	9	0.444444	8.91E-05	7.62E-07	47.22222	9
19	17	12	0.681818	4.02E-06	5.66E-08	16.41667	8
20	18	4	0	0.000127	0.000279	73.75	4
21	19	15	0.180952	0.000146	3.61E-06	11.86667	7
22	20	14	0.637363	6.24E-07	4.98E-08	14.57143	9
23	21	64	0.328373	0.007061	4.57E-05	36.54688	20
24	22	10	0.355556	5.37E-07	1.65E-06	23.4	8

node_features_structural

A1 node

	A	B	C	D	E	F	G
1	node	degree	clustering	betweenn	eigenvectc	avg_neighl	k_core
2	0	356	0.03988	0.12997	0.000351	18.99719	21
3	1	17	0.419118	1.30E-06	2.78E-06	48.76471	13
4	2	10	0.888889	0	2.29E-06	50.8	9
5	3	17	0.632353	1.17E-06	2.86E-06	60.29412	13
6	4	10	0.866667	0	2.28E-06	43.5	9
7	5	13	0.333333	3.55E-06	3.30E-06	51.30769	10
8	6	6	0.933333	0	2.23E-06	65	5
9	7	20	0.431579	5.67E-05	2.79E-05	46.35	12
10	8	8	0.678571	0	2.25E-06	49.5	5
11	9	57	0.397243	1.30E-06	5.01E-06	42.5614	21
12	10	10	0.822222	0	2.87E-06	80	10
13	11	1	0	0	2.16E-06	356	1
14	12	1	0	0	2.16E-06	356	1
15	13	31	0.651613	1.55E-06	3.53E-06	54.83871	21
16	14	15	0.742857	0	2.36E-06	39.26667	10
17	15	1	0	0	2.16E-06	356	1
18	16	9	0.666667	0	2.33E-06	67.88889	9
19	17	13	0.730769	0	2.33E-06	43.46154	9
20	18	1	0	0	2.16E-06	356	1
21	19	16	0.283333	0	2.37E-06	33.6875	7
22	20	15	0.685714	0	2.36E-06	38	9
23	21	65	0.349038	0.000651	2.89E-05	42.53846	21
24	22	11	0.472727	0	3.12E-06	54.54545	9

node_features_poison

baseline_results.txt

File Edit View

```
              precision    recall  f1-score   support

           0       0.97      1.00      0.99      1152
           1       0.88      0.50      0.64        60

    accuracy                           0.97      1212
   macro avg       0.93      0.75      0.81      1212
weighted avg       0.97      0.97      0.97      1212

Accuracy: 0.971947194719472
AUC: 0.7526475694444444
```

structural_attack_results.txt

File Edit View

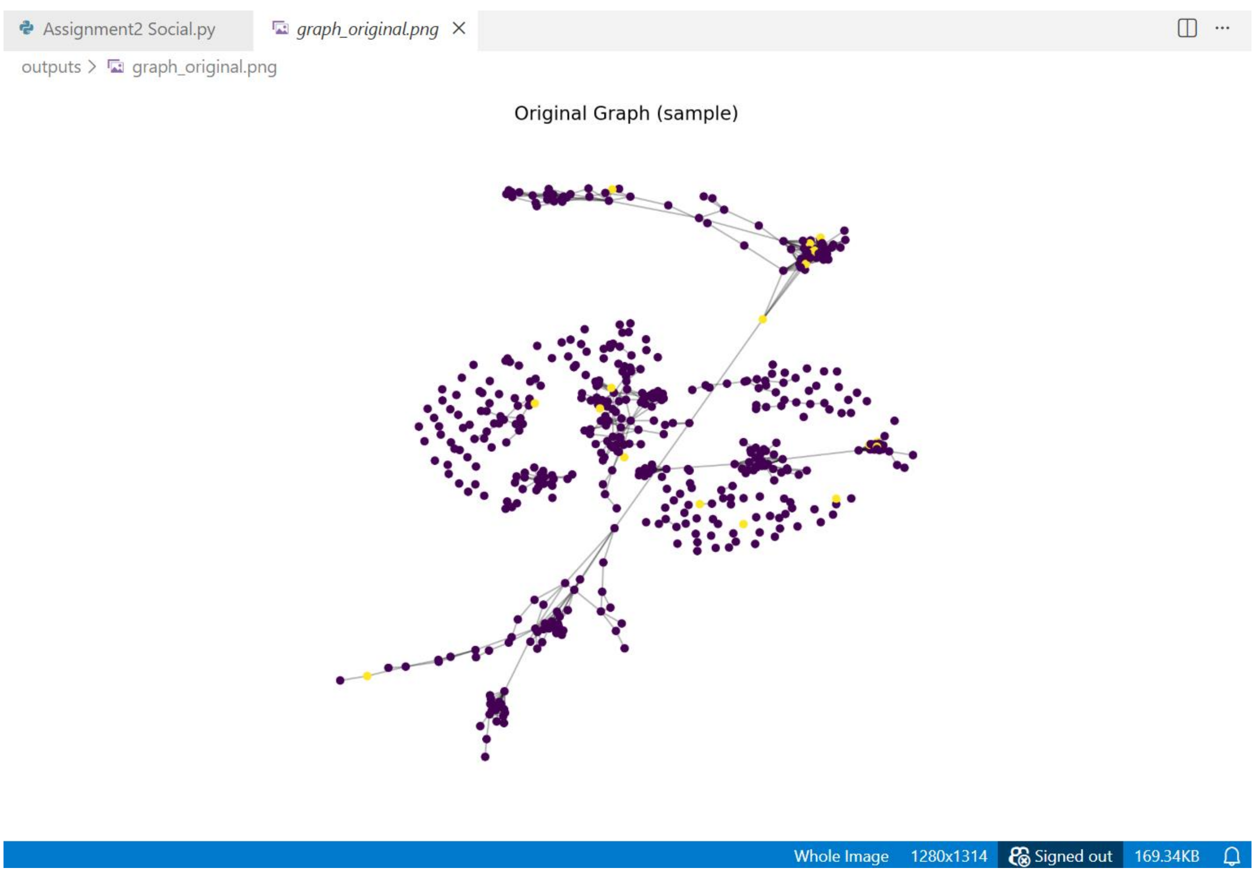
Close tab (Ctrl+W)

```
              precision    recall  f1-score   support

           0       0.96      1.00      0.98      1152
           1       1.00      0.13      0.24        60

    accuracy                           0.96      1212
   macro avg       0.98      0.57      0.61      1212
weighted avg       0.96      0.96      0.94      1212

Accuracy: 0.9570957095709571
AUC: 0.7396122685185185
```

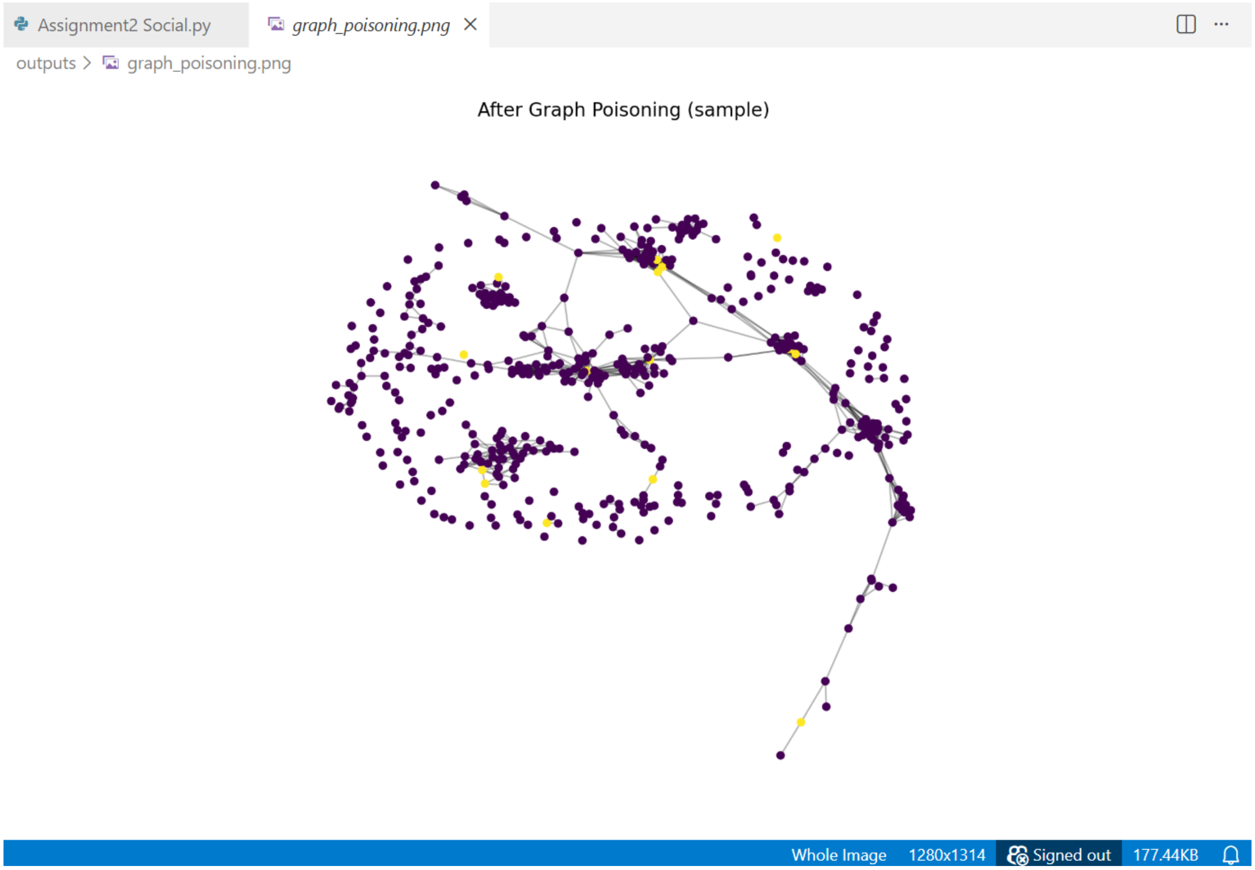
poisoning_attack_results.txt

File Edit View

```
              precision    recall  f1-score   support

           0       0.97      0.99      0.98      1152
           1       0.71      0.48      0.57        60

    accuracy                           0.96      1212
   macro avg       0.84      0.74      0.78      1212
weighted avg       0.96      0.96      0.96      1212

Accuracy: 0.9645214521452146
AUC: 0.7565972222222221
```




Assignment2 Social.py
graph_structural_attack.png
outputs > graph_structural_attack.png
After Structural Evasion (sample)
Whole Image
1280x1314
Signed out
165.41KB


Assignment2 Social.py
graph_poisoning.png
outputs > graph_poisoning.png
After Graph Poisoning (sample)
Whole Image
1280x1314
Signed out
177.44KB