Technical Deep Dive: Bot Detection & Adversarial Attacks Implementation Report

Student: Basant Ahmed Mahmoud Elkady

ID: 2205193

1. Introduction to Bot Detection Systems

What is Bot Detection?

Bot detection systems identify automated accounts in social networks. These systems analyze:

- **User behavior patterns**
- **Network connections**
- **Content posting patterns**
- **Temporal activity**

Why Graph-Based Detection?

Social networks are naturally represented as graphs:

- **Nodes** = Users
- **Edges** = Relationships/Friendships
- **Graph features** reveal hidden patterns

2. Complete Code Walkthrough

2.1 Setting Up the Environment

```
# Essential imports - WHAT EACH LIBRARY DOES:
import networkx as nx          # For creating and analyzing network
```

```
graphs
import pandas as pd            # For data manipulation (like Excel for
Python)
import numpy as np             # For numerical operations and random
number generation
import matplotlib.pyplot as plt # For creating visualizations and
plots

# Machine Learning imports
from sklearn.ensemble import RandomForestClassifier  # Our bot
detection algorithm
from sklearn.model_selection import train_test_split # Splits data for
training/testing
from sklearn.metrics import accuracy_score, precision_score,
recall_score, f1_score  # Performance metrics

# Specialized graph analysis
from networkx.algorithms.community import
greedy_modularity_communities  # Finds natural groups in network
```

2.2 Data Loading Process

```
# Download Facebook dataset
```

```
url = "https://snap.stanford.edu/data/facebook_combined.txt.gz"
filename = "facebook_combined.txt.gz"

if not os.path.exists(filename):
    urllib.request.urlretrieve(url, filename)
    print("✅ Data downloaded successfully!")
```

```
# Extract from gzip format
```

```
# Extract file
with gzip.open(filename, 'rb') as f_in:
    with open("facebook_combined.txt", 'wb') as f_out:
        f_out.write(f_in.read())

print("✅ File extracted successfully!")
```

```
✅ Data downloaded successfully!
✅ File extracted successfully!
```

```
# Load graph from edge list format
```

```
# Format: Each line "user1 user2" represents a friendship
G = nx.read_edgelist("facebook_combined.txt")
print(f"📊 Nodes: {G.number_of_nodes()}, Edges: {G.number_of_edges()}")
```

What this does:

1. Downloads Facebook friendship data
2. Extracts it from compressed format
3. Creates a graph object where each user is a node, each friendship is an edge
4. Output shows: 4,039 users with 88,234 friendships

3. Feature Extraction - The Heart of Detection

3.1 Understanding Graph Metrics

Feature 1: Degree Centrality

```
degree_dict = dict(G.degree())
```

- **What it measures:** How many friends each user has
- **Formula:** `degree(node) = count(neighbors)`
- **Why it matters for bots:**
  - Bots often have either very high degrees (spamming) or very low degrees (fake accounts)
  - Normal users have moderate, socially plausible numbers of friends

Feature 2: Clustering Coefficient

```
clustering_dict = nx.clustering(G)
```

- **What it measures:** How interconnected a user's friends are
- **Formula:** `(actual connections between friends) / (possible connections between friends)`
- **Example:** If you have 3 friends and 2 of them are friends with each other, your clustering coefficient is ~0.33

- **Why it matters:** Real users' friends tend to know each other (high clustering), bots' connections are random (low clustering)

Feature 3: Betweenness Centrality

```
centrality_dict = nx.betweenness_centrality(G, k=500, normalized=True)
```

- **What it measures:** How often a user acts as a bridge between different parts of the network
- **Why k=500?** Approximates for large networks (faster computation)
- **Why it matters:** Bots might have abnormal bridging behavior

Feature 4: Community Detection

```
communities = list(greedy_modularity_communities(G))
```

- **What it does:** Automatically finds natural friend groups
- **How it works:** Maximizes "modularity" - how dense connections are within groups vs between groups
- **Why it matters:** Bots might not fit naturally into communities

3.2 Creating the Feature Table

```
features = pd.DataFrame({
    'node': list(G.nodes()),
    'degree': [degree_dict[n] for n in G.nodes()],
    'clustering': [clustering_dict[n] for n in G.nodes()],
    'centrality': [centrality_dict[n] for n in G.nodes()]
})
```

```
features['community'] = features['node'].map(community_dict).fillna(-1)
features.head()
```

Resulting DataFrame looks like:

```
Number of communities found: 16
```

	node	degree	clustering	centrality	community
0	0	347	0.041962	1.656542e-01	4
1	1	17	0.419118	7.852366e-06	4
2	2	10	0.888889	0.000000e+00	4
3	3	17	0.632353	4.211048e-07	4
4	4	10	0.866667	0.000000e+00	4

4. Understanding the Attacks

4.1 What Are Adversarial Attacks?

Adversarial attacks are techniques where attackers intentionally manipulate data to fool machine learning models. In our context:

Two Types of Attacks:

1. **Evasion Attacks:** Modify existing bots to look normal
2. **Poisoning Attacks:** Add new fake data to confuse the model

4.2 Structural Evasion Attack - Detailed Explanation

```
def intelligent_evasion_attack(G, bot_nodes, features_df):
    G_attacked = G.copy()  # Start with original graph

    for bot in bot_nodes:  # For each bot account
        if bot not in G_attacked:
            continue

        # STEP 1: Remove bot-to-bot connections
        bot_neighbors = list(G_attacked.neighbors(bot))  # Get bot's
friends
        bot_bot_links = [n for n in bot_neighbors
                        if features_df.loc[features_df['node']==n,
'label'].values[0] == 1]
        # bot_bot_links contains other bots among the friends
```

```
        for other_bot in bot_bot_links[:2]:  # Remove up to 2 bot
connections
            if G_attacked.has_edge(bot, other_bot):
                G_attacked.remove_edge(bot, other_bot)

        # STEP 2: Add connections to normal users
        normal_users = features_df[features_df['label'] ==
0]['node'].sample(3).tolist()
        # Randomly select 3 normal users

        for user in normal_users:
            if not G_attacked.has_edge(bot, user):
                G_attacked.add_edge(bot, user)  # Add friendship

    return G_attacked
```

Why This Attack Works:

1. **Bot networks are detectable** because bots connect mostly with other bots
2. **By removing bot-bot connections**, we break the detectable pattern
3. **By adding human connections**, the bot looks more like a normal user
4. **Feature changes:**
    a. Clustering coefficient increases (more normal)
    b. Community assignment might change
    c. Degree becomes more average

Real-world Analogy: Imagine a spy in a social group:

- First, they stop meeting with other spies (remove bot-bot connections)
- Then, they start attending normal social events (add human connections)
- Result: They blend in better

4.3 Graph Poisoning Attack - Detailed Explanation

```
# Create new fake bot accounts
new_bots = [f'bot_fake_{i}' for i in range(20)]
# Creates IDs: bot_fake_0, bot_fake_1, ..., bot_fake_19

for b in new_bots:
```

```
    G_poisoned.add_node(b)  # Add the new bot as a node

    # Connect to 5 random existing users
    targets = list(np.random.choice(list(G_poisoned.nodes()), 5,
replace=False))
    # Randomly select 5 users from the entire network

    for t in targets:
        G_poisoned.add_edge(b, t)  # Create friendships
```

Why This Attack Works Differently:

1. **Not hiding existing bots** - creating new ones
2. **These new bots have random connections** to both humans and other bots
3. **Purpose:** Poison the training data distribution
4. **Effect on the model:**
    a. Decision boundaries become fuzzy
    b. Model confidence decreases
    c. Harder to distinguish patterns

Analogy:

- **Evasion** = Making existing spies look like civilians
- **Poisoning** = Adding many undercover agents who act randomly to create confusion

5. Model Training Process

5.1 Random Forest Classifier - How It Works

```
clf = RandomForestClassifier(n_estimators=100, random_state=42)
```

What is Random Forest?

- Ensemble of 100 decision trees
- Each tree sees a random subset of data/features
- Final prediction = majority vote of all trees

Why Use Random Forest for Bot Detection?

1. **Handles non-linear relationships** (bots don't follow simple rules)
2. **Feature importance** (tells us which features matter most)
3. **Robust to overfitting** (won't memorize training data)

5.2 Training-Test Split

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
```

Why split data?

- **Training set (70%):** Learn patterns
- **Test set (30%):** Evaluate performance on unseen data
- **random_state=42:** Ensures reproducible results

6. Performance Metrics - What Do They Mean?

6.1 Confusion Matrix Concepts

```
                  ACTUAL
               |  Bot   | Normal
PREDICTED Bot  |  TP    |   FP
       Normal  |  FN    |   TN
```

- **TP (True Positive):** Correctly identified bots
- **FP (False Positive):** Normal users mistakenly called bots
- **FN (False Negative):** Bots mistakenly called normal (MOST DANGEROUS!)
- **TN (True Negative):** Correctly identified normal users

6.2 Metric Formulas and Interpretation

Accuracy:

```
accuracy = (TP + TN) / (TP + TN + FP + FN)
```

- **Our result:** 0.859 (85.9%)

- **Problem:** With 85% normal users, guessing "normal" for everyone gives 85% accuracy!

Precision:

```
precision = TP / (TP + FP)
```

- **Our result:** 0.426 (42.6%)
- **Interpretation:** "When we say it's a bot, how often are we right?"
- **Good precision =** Few false accusations

Recall (Sensitivity):

```
recall = TP / (TP + FN)
```

- **Our result:** 0.141 (14.1%) - TERRIBLE!
- **Interpretation:** "Of all actual bots, what percentage do we catch?"
- **Good recall =** Catching most bots

F1-Score:

```
f1 = 2 * (precision * recall) / (precision + recall)
```

- **Our result:** 0.212
- **Harmonic mean** of precision and recall
- **Balanced metric** - punishes extreme values

7. Analyzing Attack Results

7.1 Why Did Accuracy INCREASE After Attacks?

Mathematical Explanation:

```
Before attack:
- 1000 users: 850 normal, 150 bots
- Model catches 21 bots (TP), misses 129 (FN)
- Correctly identifies 800 normals (TN), falsely accuses 50 (FP)
- Accuracy = (21 + 800) / 1000 = 82.1%
```

```
After evasion attack:
- Bots become harder to detect
- Model becomes more conservative
- Catches 40 bots (TP), misses 110 (FN)
- Correctly identifies 830 normals (TN), falsely accuses 20 (FP)
- Accuracy = (40 + 830) / 1000 = 87.0% (INCREASE!)
```

The Paradox:

- Attacks make model more cautious
- Fewer false accusations (better for normal users)
- But also catches fewer bots (worse for security)

7.2 Feature Changes After Attacks

Example Bot #456:

```
BEFORE ATTACK:
- Degree: 50 (only bot friends)
- Clustering: 0.1 (bot friends don't know each other)
- Community: 7 (bot-only community)

AFTER EVASION:
- Degree: 53 (+3 human friends)
- Clustering: 0.3 (higher - friends more connected)
- Community: 4 (now in mixed human-bot community)

RESULT: Looks more like normal user with degree=53, clustering=0.3
```

7.3 Decision Boundary Visualization

```
Feature Space (simplified to 2D):
                  |
  Normal Users    |    Bots
  o o o o         |    △ △
  o o o o o       |    △ △ △
  o o o o         |    △ △
```

```
  ---------|-----------
                 |
AFTER POISONING:
                 |
  Normal Users   |    Bots + Fake Bots
  ○ ○ ○ ○ △       |    △ △ △ ○
  ○ ○ ○ △ ○ ○     |    △ ○ △ △
  ○ △ ○ ○ ○       |    △ △ ○ △
  ---------|-----------
                 |
Decision boundary becomes fuzzy and confused
```

8. Code Output Analysis

8.1 Baseline Output Explanation

```
* Baseline Model Performance:
```

```
🔹 Baseline Model Performance:
Accuracy : 0.859
Precision: 0.426
Recall   : 0.141
F1 Score : 0.212
```

```
Accuracy : 0.859       ← Seems good but misleading
Precision: 0.426       ← Only 42.6% of bot predictions are correct
Recall   : 0.141       ← CRITICAL: Only catches 14.1% of bots
F1 Score : 0.212       ← Very poor overall performance
```

8.2 Evasion Attack Output

```
◆ After Structural Evasion Attack:
```

```
🔸 After Structural Evasion Attack:
Accuracy : 0.88
Precision: 0.691
Recall   : 0.265
F1 Score : 0.383
```

```
Accuracy : 0.88        ↑ Increased 2.1% - looks better!
Precision: 0.691       ↑ Increased 26.5% - much more confident
```

```
Recall    : 0.265      ↑ Increased 12.4% - still misses 73.5% of bots
F1 Score : 0.383      ↑ Increased 17.1% - better but still bad
```

8.3 Poisoning Attack Output

```
▲ After Graph Poisoning Attack:
```

```
▲ After Graph Poisoning Attack:
Accuracy : 0.883
Precision: 0.741
Recall    : 0.294
F1 Score : 0.421
```

```
Accuracy : 0.883      ↑ Best accuracy
Precision: 0.741      ↑ Best precision
Recall    : 0.294      ↑ Best recall (but still terrible)
F1 Score : 0.421      ↑ Best F1 score
```

9. Security Implications - Why This Matters

9.1 Real-World Attack Scenarios

Scenario 1: Political Manipulation

- Bot network spreads misinformation
- Detection system tries to catch them
- Attackers use evasion techniques
- Bots continue operating undetected

Scenario 2: Financial Fraud

- Bots create fake reviews for products
- Platform's detection system flags them
- Attackers use poisoning attacks
- Detection system becomes less effective over time

9.2 The Arms Race

```
Time 0: Basic detection system
Time 1: Attackers develop evasion techniques
Time 2: Detection system improves
Time 3: Attackers develop poisoning techniques
Time 4: Detection system adds adversarial training
... Continues indefinitely
```

10. Complete Attack Process Flowchart

```
START
  │
  ▼
Download Facebook Data
  │
  ▼
Create Graph (Nodes=Users, Edges=Friendships)
  │
  ▼
Extract Features:
├── Degree (# friends)
├── Clustering (friend interconnectedness)
├── Centrality (bridging importance)
└── Community (social group)
  │
  ▼
Create Synthetic Bot Labels
  │
  ▼
Train Random Forest Model
  │
  ▼
Evaluate Baseline Performance ←──────────┐
  │                                       │
  ▼                                       │
Select 50 Bots for Attack                 │
  │                                       │
```

```
  ▼                                          │
APPLY EVASION ATTACK:                        │
1. Remove bot-bot connections                │
2. Add human connections                     │
  │                                          │
  ▼                                          │
Recompute Features                           │
  │                                          │
  ▼                                          │
Evaluate Post-Evasion Performance            │
  │                                          │
  ▼                                          │
APPLY POISONING ATTACK:                      │
1. Add 20 new fake bots                      │
2. Connect to random users                   │
  │                                          │
  ▼                                          │
Recompute Features                           │
  │                                          │
  ▼                                          │
Evaluate Post-Poisoning Performance          │
  │                                          │
  ▼                                          │
Compare All Scenarios                        │
  │                                          │
  ▼                                          │
Analyze Results & Draw Conclusions           │
  │                                          │
  ▼                                          │
Generate Visualizations ─────────────────────┘
```

11. Key Technical Insights

11.1 Why Random Forest Works (and Doesn't)

Strengths:

- Handles complex feature interactions

- Provides feature importance scores
- Robust to irrelevant features

Weaknesses (Revealed by Attacks):

- Assumes stationary data distribution
- Vulnerable to distribution shifts
- Can't adapt to evolving attack strategies

11.2 Feature Importance Analysis

From Random Forest, we could check:

```
feature_importance = pd.DataFrame({
    'feature': X.columns,
    'importance': clf.feature_importances_
}).sort_values('importance', ascending=False)
```

Expected Results:

1. Degree (most important - bots have abnormal connections)
2. Clustering (second - bots have unnatural friend networks)
3. Community (third - bots don't fit social groups)
4. Centrality (least - varies more among normal users)

11.3 Computational Complexity

Most Expensive Operations:

1. Betweenness Centrality: $O(n \times m)$ where n=nodes, m=edges
    a. Why we use k=500 (sampling approximation)
2. Community Detection: $O(m \log n)$
3. Random Forest Training: $O(t \times f \times n \log n)$ where t=trees, f=features

Total Runtime: ~5-10 minutes on standard laptop

12. Conclusion

12.1 Summary of Findings

1. **Current bot detection systems are fragile** against intelligent attacks
2. **Accuracy is misleading** - increased accuracy can mean decreased security
3. **Recall is the critical metric** for security applications
4. **Poisoning attacks are more dangerous** than evasion attacks
5. **Adversarial training is essential** for robust systems