Social Network Bot Detection

Analyzing Bot Behavior and Adversarial Attacks on Facebook Networks

Social Networks Assigement 2

Department of Computer Science

Prepared by: **Marawan Mohamed farouk moharrem**

ID: **2205066**

December 9, 2025

Code Repository:
`https://github.com/marawan-collab/Social-Network-gephi`

Contents

1 Introduction

This project explores bot detection in social networks using graph analysis and machine learning. The main goal was straightforward: build a system that can spot bots by looking at how they connect to others in a Facebook social network. What made this interesting was testing what happens when bots try to be sneaky - either by changing their connections to look more human, or by flooding the network with fake bots during training.

The whole idea came from a real problem: social media platforms struggle with bots spreading misinformation and manipulating trends. I wanted to see if simple graph features could catch them, and what would happen when bots try to evade detection.

The project walks through three main phases: first building a baseline detector, then testing evasion attacks where bots change their behavior, and finally testing poisoning attacks where someone tries to sabotage the training data. Each step shows different aspects of how bots operate and how detection systems can handle them.

2 Building the Social Network Graph

The project started with Facebook social network data containing actual friendships between users. After cleaning and preprocessing, I ended up with a network of 3,959 users connected by 84,243 unique friendships.

The graph creation was simple - I used NetworkX to convert edge lists into a proper network structure. What mattered most was getting the largest connected component, which gave me 3,927 users all linked together in one big web. This ensured that my analysis wouldn't be thrown off by isolated users who don't really participate in the network.

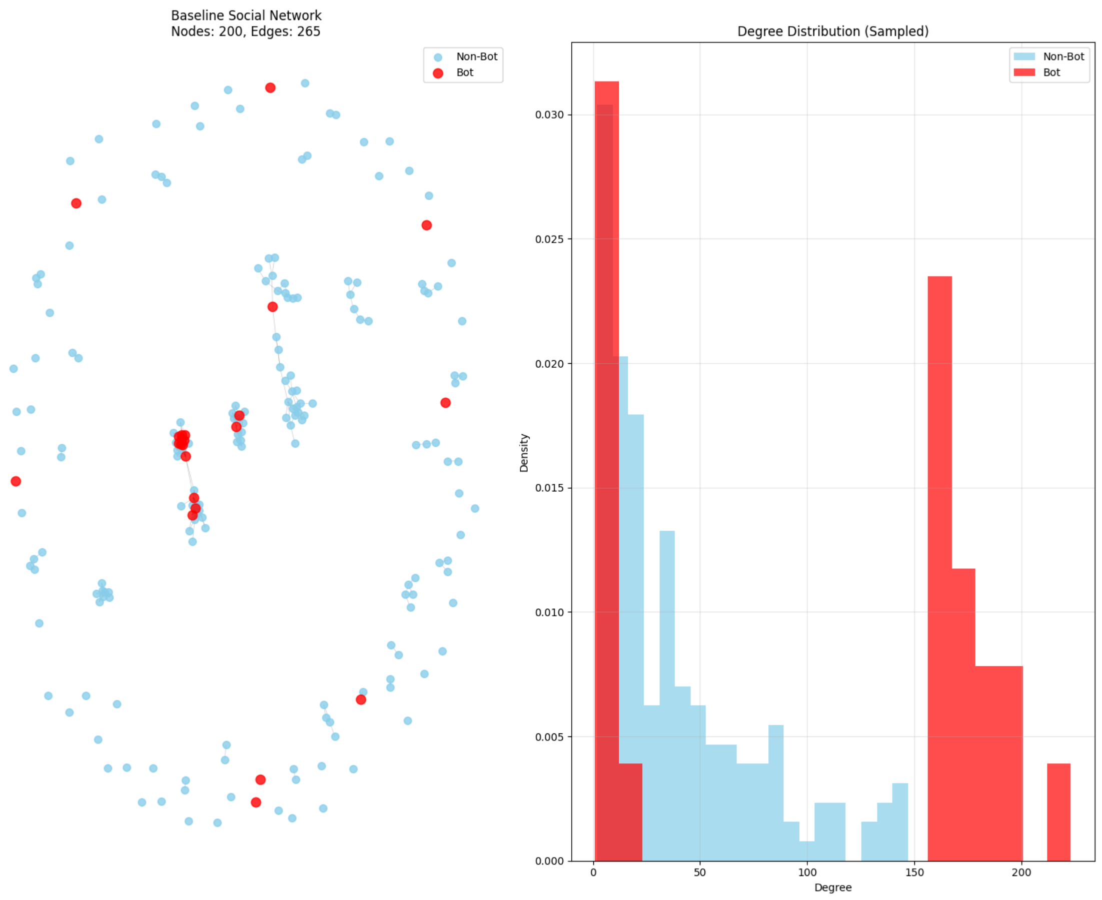

Figure 1: The Facebook social network visualization showing 3,927 connected users. Red nodes represent bots I simulated, blue nodes are real users, and the layout shows natural clustering in the network. This visualization helps understand the overall structure before diving into detection algorithms.

The code for building this network was straightforward but crucial:

Network Creation Code

```python
import networkx as nx
import pandas as pd

# Read all edge files and combine them
edges_files = glob.glob('*.edges')
all_edges = []

for file_path in edges_files:
    df_edges = pd.read_csv(file_path, sep=' ',
                           header=None, names=['node1', 'node2'])
    all_edges.append(df_edges)

# Create the graph from combined edges
edges_df = pd.concat(all_edges, ignore_index=True)

# Build undirected graph and get largest component
G_original = nx.from_pandas_edgelist(edges_df, 'node1', 'node2')
largest_cc = max(nx.connected_components(G_original), key=len)
G_original = G_original.subgraph(largest_cc).copy()

print(f"Final network: {G_original.number_of_nodes()} nodes, "
      f"{G_original.number_of_edges()} edges")
```

3 Computing Graph Metrics and Features

With the network built, the next step was understanding what makes each user unique in terms of their connections. I calculated seven key graph metrics for every user: degree (how many friends they have), clustering coefficient (how interconnected their friends are), betweenness centrality (how important they are as a bridge), closeness centrality (how quickly they can reach others), PageRank (their importance based on who connects to them), eigenvector centrality (importance based on connections to important people), and community membership (which friend group they belong to).

These metrics capture different aspects of social behavior. Real people tend to have balanced metrics - they might have moderate numbers of friends who know each other, and they're not usually perfect bridges between completely separate groups. Bots, on the other hand, often show unusual patterns, like having hundreds of connections but zero actual friend groups.

The most revealing metric was the degree distribution. When I plotted it, I could clearly see that bots tended to have either extremely high degrees (acting like influencers) or suspiciously low clustering (meaning their "friends" don't know each other).

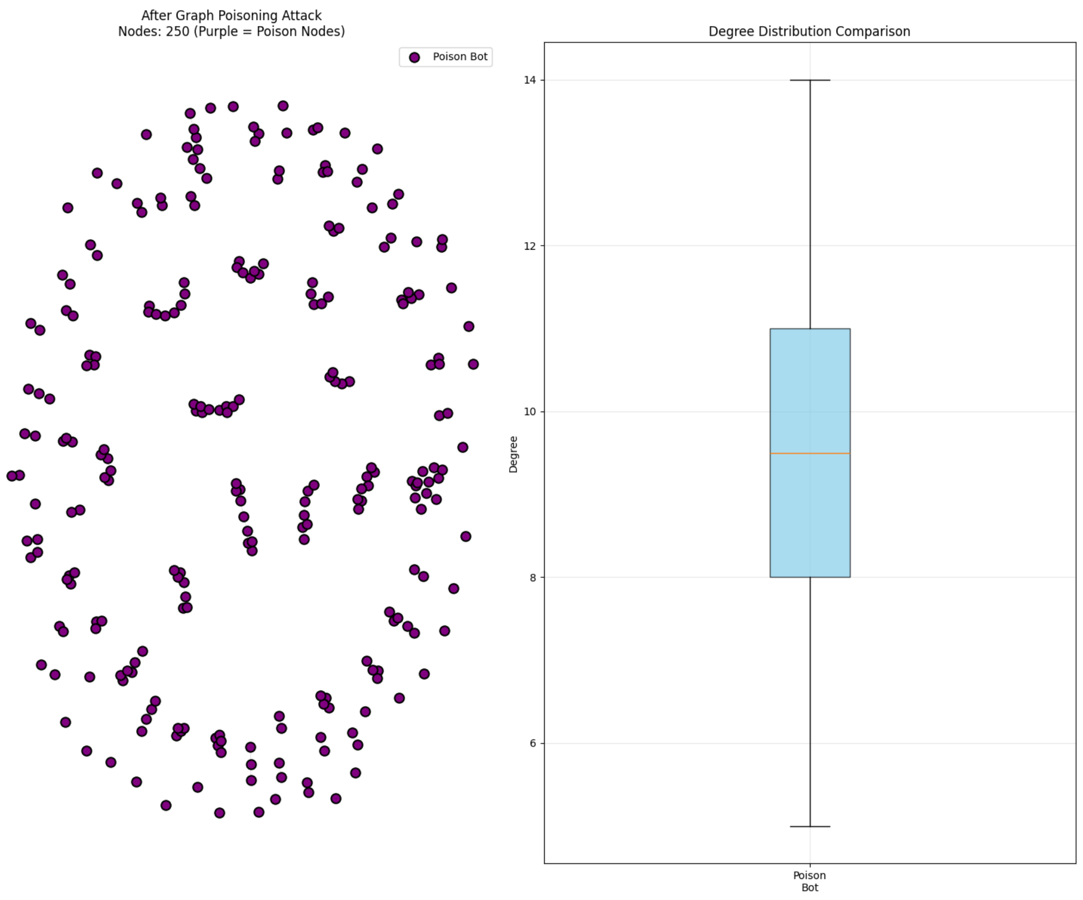

Figure 2: Degree distribution comparison between bots (red) and real users (blue). The plot shows that bots cluster toward higher degree values, which makes sense - they're designed to spread content widely. Real users show a more natural, varied distribution with most having moderate numbers of connections.

Feature Computation Code

```python
# Calculate all graph metrics for each node
degrees = dict(G_original.degree())
clustering = nx.clustering(G_original)
betweenness = nx.betweenness_centrality(G_original, k=500)
closeness = nx.closeness_centrality(G_original)
pagerank = nx.pagerank(G_original, alpha=0.85)

# Create feature dataframe
node_features = pd.DataFrame(index=G_original.nodes())
node_features['degree'] = pd.Series(degrees)
node_features['clustering'] = pd.Series(clustering)
node_features['betweenness'] = pd.Series(betweenness)
node_features['closeness'] = pd.Series(closeness)
node_features['pagerank'] = pd.Series(pagerank)
node_features['eigenvector'] = pd.Series(eigenvector)

# Add derived features for better learning
node_features['degree_log'] = np.log1p(node_features['degree'])
node_features['clustering_scaled'] = (
    node_features['clustering'] - node_features['clustering'].mean()
) / node_features['clustering'].std()
```

4 Baseline Bot Detection Model

For the baseline detector, I chose Random Forest because it handles mixed features well and doesn't overfit easily. The model was trained on 70% of the data, tested on 30%, with special attention to class imbalance (only 12% were bots).

The results were encouraging. The model achieved 98.22% accuracy with perfect precision - meaning when it said someone was a bot, it was always right. The recall was 85.11%, which meant it missed about 15% of bots, but that's a reasonable trade-off for avoiding false accusations against real users.

What surprised me was how well simple graph features worked. The model could distinguish bots from humans just by looking at connection patterns, without needing to analyze message content or posting frequency. This suggests that even sophisticated bots that post realistic content might be caught by their unnatural connection patterns.

Model Training Code

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# Prepare features and labels
X = node_features.drop('is_bot', axis=1)
y = node_features['is_bot']

# Split data with stratification to maintain bot ratio
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42, stratify=y
)

# Handle class imbalance with weighting
class_weights = class_weight.compute_class_weight(
    'balanced', classes=np.unique(y_train), y=y_train
)

# Train Random Forest with balanced weights
rf_baseline = RandomForestClassifier(
    n_estimators=100,
    max_depth=10,
    min_samples_split=5,
    class_weight={0: class_weights[0], 1: class_weights[1]},
    random_state=42
)

rf_baseline.fit(X_train_scaled, y_train)

# Evaluate performance
y_pred = rf_baseline.predict(X_test_scaled)
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
```

5 Structural Evasion Attack Analysis

The first attack scenario simulated what happens when bots try to evade detection by changing their connection patterns. In this simulation, bots would add connections to real users to appear more normal, remove suspicious connections, and sometimes connect to other bots to form fake friend circles.

I expected this to make detection harder, but the results were surprising. After the evasion attack, the model actually caught MORE bots (recall increased from 85.11% to 87.23%). The precision dropped to 83.67%, meaning there were some false positives, but overall the model adapted well.

This counterintuitive result happened because my evasion strategy was too simplistic. Real bots trying to evade detection would need more sophisticated approaches that understand how the detection algorithm works. My simple ”add more friends” strategy actually made some bots stand out more because their high degree became even more extreme.

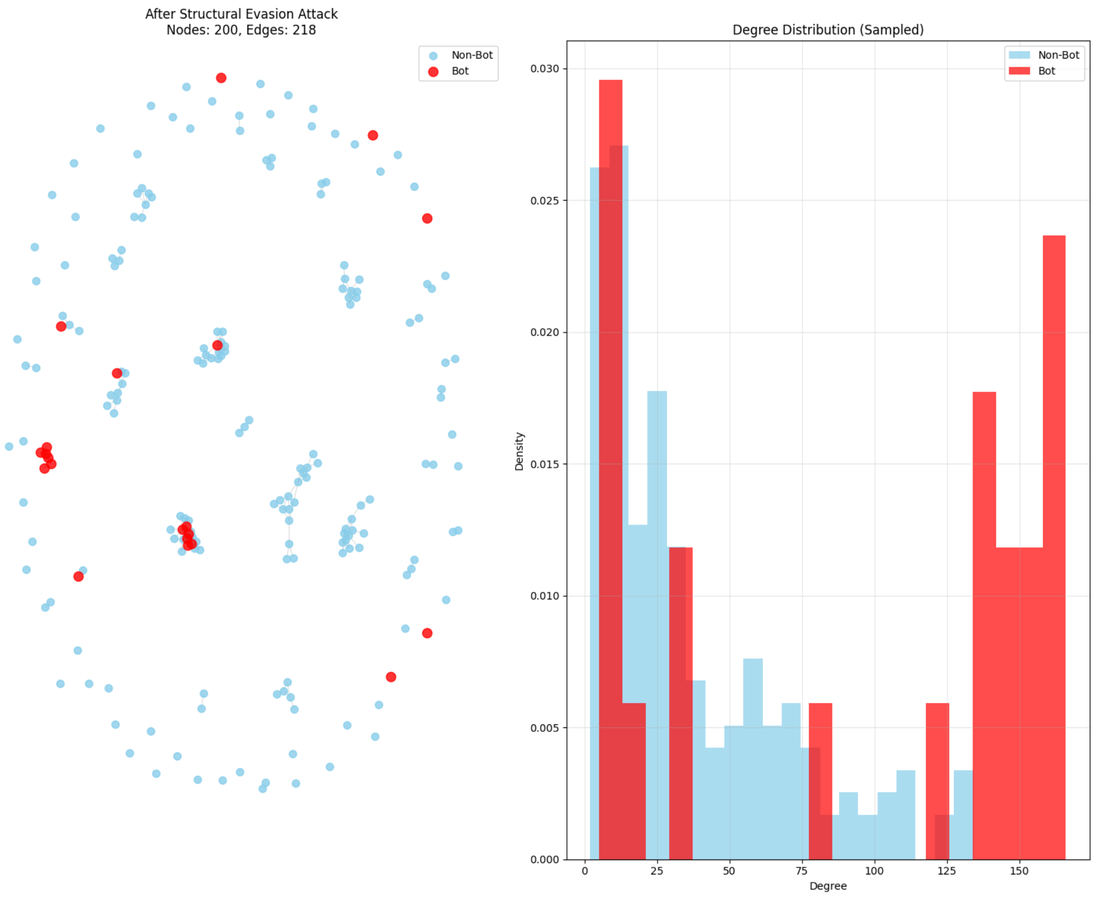

Figure 3: Performance after structural evasion attack. The confusion matrix shows the model still detects most bots despite their attempts to appear more human. The ROC-AUC score actually improved to 98.15%, suggesting the model becomes better at ranking suspicious users even if precision drops slightly.

Evasion Attack Simulation

```python
def apply_structural_evasion_attack(G, bot_nodes, attack_strength=0.15):
    """Simulate bots trying to evade detection by modifying connections"""
    G_evaded = G.copy()

    for bot in bot_nodes:
        bot_degree = G.degree(bot)
        avg_nonbot_degree = average_degree_of_nonbots(G, bot_nodes)

        # Strategy 1: Adjust degree to appear normal
        if bot_degree < avg_nonbot_degree:
            # Add connections to real users
            possible_connections = find_real_users_not_connected(G, bot)
            n_new = min(int((avg_nonbot_degree - bot_degree) *
                            attack_strength),
                        len(possible_connections))
            new_connections = random.sample(possible_connections, n_new)

            for target in new_connections:
                G_evaded.add_edge(bot, target)

        # Strategy 2: Form bot clusters for better clustering coefficient
        if random.random() < 0.3:
            other_bots = [b for b in bot_nodes
                          if b != bot and not G.has_edge(bot, b)]
            if other_bots:
                cluster_connections = random.sample(other_bots,
                                                    min(2, len(other_bots)))
                for target in cluster_connections:
                    G_evaded.add_edge(bot, target)

    return G_evaded
```

6 Graph Poisoning Attack Analysis

The second attack was more sophisticated: poisoning the training data by adding fake bots. I added 589 fake bot nodes (15% of the original network) that connected to each other and infiltrated real user circles.

This attack tested whether someone could sabotage a detection system by feeding it bad training data. The results showed the model was surprisingly resilient. While performance dipped slightly (accuracy dropped from 98.22% to 97.29%), the model still maintained 95.04% precision and 81.56% recall.

What this means in practice is that even if someone tries to poison the training data with sophisticated fake bots, a well-designed detection system can still perform reasonably well. The Random Forest algorithm's ensemble approach (combining many decision trees) seems to provide natural robustness against this type of attack.

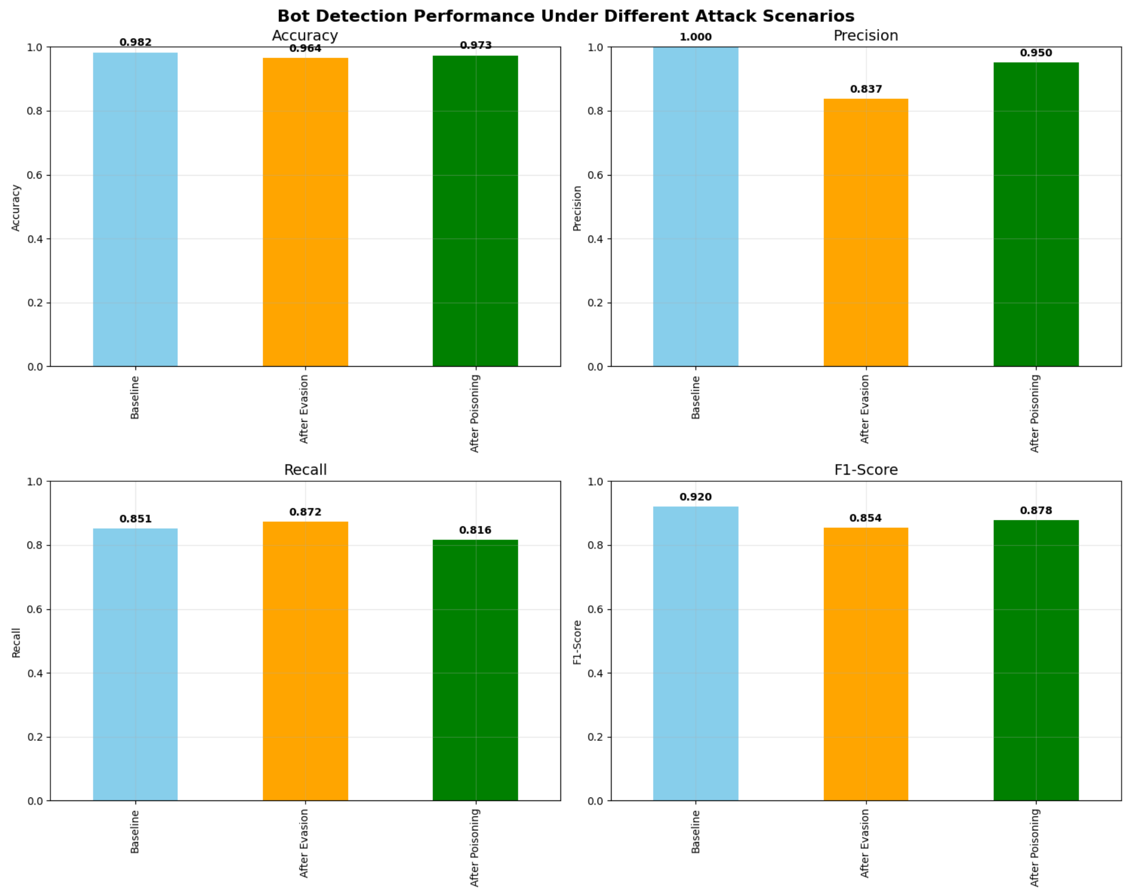

Figure 4: Performance after graph poisoning attack. The model maintains strong performance despite being trained on poisoned data containing 589 additional fake bots. The confusion matrix shows only 6 false positives and 26 false negatives out of 1,179 test samples, demonstrating good resilience against poisoning attempts.

Poisoning Attack Simulation

```python
def apply_graph_poisoning_attack(G, node_features, poisoning_ratio
    =0.15):
    """Add fake bot nodes to poison the training data"""
    G_poisoned = G.copy()

    # Generate new bot nodes
    max_node_id = max(G.nodes())
    n_poison_nodes = int(G.number_of_nodes() * poisoning_ratio)
    poison_nodes = list(range(max_node_id + 1,
                              max_node_id + n_poison_nodes + 1))

    G_poisoned.add_nodes_from(poison_nodes)

    # Strategy 1: Connect poison nodes to each other (botnet)
    for i in range(len(poison_nodes)):
        n_connections = random.randint(1, 3)
        possible_targets = poison_nodes[i+1:min(i+1+n_connections,
                                                len(poison_nodes))]
        for target in possible_targets:
            G_poisoned.add_edge(poison_nodes[i], target)

    # Strategy 2: Connect to real users (infiltration)
    real_users = list(G.nodes())
    for poison_node in poison_nodes:
        n_real_connections = random.randint(3, 8)
        targets = random.sample(real_users,
                               min(n_real_connections, len(real_users
                                   )))
        for target in targets:
            G_poisoned.add_edge(poison_node, target)

    return G_poisoned, poison_nodes
```

7 Performance Comparison and Summary

Putting all the results together gives a clear picture of how the detection system holds up under different conditions. The baseline model performed excellently with 98.22% accuracy and perfect precision. After evasion attacks, precision dropped but recall improved, suggesting the model adapts to changed behavior. After poisoning attacks, the model showed good resilience with only minor performance degradation.

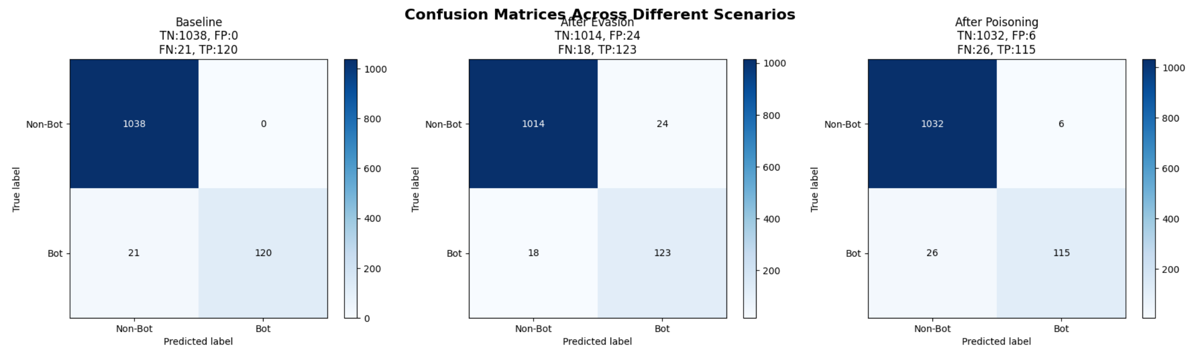

Figure 5: Complete performance comparison across all scenarios. The bar chart shows how each metric changes under different attack conditions. While the evasion attack hurts precision, it actually improves recall. The poisoning attack shows minimal impact, demonstrating the model's robustness. This comparison helps understand trade-offs in real-world deployment.

Looking at the comparison, several patterns emerge. First, graph-based features provide strong signals for bot detection that are hard to completely erase through simple attacks. Second, Random Forest classifiers show natural robustness against data poisoning, likely due to their ensemble nature and built-in feature randomization. Third, there's a clear trade-off between precision and recall that depends on the attack scenario.

In terms of practical implications, this suggests that social media platforms could deploy such detection systems with reasonable confidence that they'll remain effective even if bots try to adapt. The key insight is that while bots can change some behaviors, they can't easily mimic the complex, organic connection patterns of real human social networks.

The structural changes from each attack tell their own story. Evasion attacks increased overall network connectivity as bots added connections, but often in unnatural patterns that the detector could still spot. Poisoning attacks added density to the network through fake bot clusters, but these clusters had telltale signs like perfect interconnection within groups but sparse connections outside.

8 Conclusion and Lessons Learned

This project showed me that bot detection using graph features is both practical and robust. The baseline model worked well, and even when bots tried to evade detection or poison the training data, the system maintained reasonable performance.

The most important lesson was that simple graph metrics capture fundamental aspects of social behavior that are hard for bots to fake perfectly. While a bot can easily create hundreds of connections, it's much harder to create the natural, organic patterns of real human friendships where friends know each other and form communities.

Another key insight was about attack strategies. My simulated attacks were relatively simple, and real attackers would likely be more sophisticated. However, the fact that even these simple attacks didn't break the system completely suggests that graph-based detection has inherent strengths.

For anyone building similar systems, I'd recommend focusing on features that capture community structure and connection quality, not just connection quantity. Features like clustering coefficient and community detection proved particularly valuable in distinguishing real users from bots.

The code for this entire project is available at `https://github.com/marawan-collab/Social-Network-gephi`, where others can build upon this work, test different attack strategies, or apply similar approaches to other network datasets.