Signed Networks

Introduction

Signed networks are graphs where edges can have positive (+) or negative (-) weights, representing relationships such as friendship/enmity, trust/distrust, or agreement/disagreement. These networks provide a richer representation of social and economic interactions than traditional unsigned networks.

Basic Concepts

1. Signed Network Structure

Definition: A signed network is a graph $G = (V, E, \sigma)$ where:

- V is the set of nodes
- E is the set of edges
- $\sigma : E \to \{+1, -1\}$ is the sign function

Edge Types:

- **Positive edges** $(+1)$: Friendship, trust, agreement
- **Negative edges** (-1): Enmity, distrust, disagreement

2. Adjacency Matrix Representation

Signed Adjacency Matrix:

$$A_{ij} = \begin{cases} +1 & \text{if } (i, j) \text{ is a positive edge} \\ -1 & \text{if } (i, j) \text{ is a negative edge} \\ 0 & \text{if no edge exists} \end{cases}$$

Balance Theory

1. Structural Balance

Definition: A signed network is balanced if it can be partitioned into two groups such that all positive edges are within groups and all negative edges are between groups.

Mathematical Definition: A triangle is balanced if the product of its edge signs is positive:

$$\sigma_{ij} \cdot \sigma_{jk} \cdot \sigma_{ki} = +1$$

2. Balance Conditions

Triangle Types:

1. **+++**: All positive edges (balanced) - "Friend of friend is friend"
2. **+--**: One positive, two negative edges (balanced) - "Enemy of enemy is friend"
3. **++-**: Two positive, one negative edge (unbalanced) - "Two friends with common enemy"
4. **---**: All negative edges (unbalanced) - "Three mutual enemies"

3. Balance Index

Definition: Fraction of balanced triangles in the network.

Mathematical Definition:

$$B = \frac{\text{Number of balanced triangles}}{\text{Total number of triangles}}$$

Python Implementation: ` ` `python def calculate_balance_index(G): """Calculate balance index for signed network""" triangles = [c for c in nx.enumerate_all_cliques(G) if len(c) == 3]

```
if not triangles:
    return None

balanced = 0
for triangle in triangles:
```

```
    a, b, c = triangle
    # Product of three edge signs
    product = (G[a][b]['sign'] * G[b][c]['sign'] *
               G[c][a]['sign'])
    if product == 1:
        balanced += 1

return balanced / len(triangles)
```

```

## Centrality Measures

> ⚠️ ***CRITICAL WARNING****: Standard centrality measures (betweenness, closeness, eigenvector) were designed for unsigned networks and* ***DO NOT work properly for signed networks!*** *Always use specialized signed network methods.*

### Issues with Standard Centrality

**Problems**:

1. **Shortest Paths**: How to define "shortest" when edges have different signs?
2. **Path Meaning**: A path through enemies is fundamentally different from a path through friends
3. **Edge Treatment**: Standard algorithms treat all edges equally, ignoring signs
4. **Existence**: Some measures (like eigenvector centrality) may not even exist for signed networks!

**Example of Failure**: `python`

```
# X WRONG - These ignore edge signs completely!
betweenness = nx.betweenness_centrality(G) # Treats all edges equally
closeness = nx.closeness_centrality(G)   # Ignores sign meaning
eigenvector = nx.eigenvector_centrality(G) # May not exist!
```

---
```

1. Signed Degree Centrality

Definition: Net degree considering both positive and negative connections.

Mathematical Definition:

$$d_i^{net} = d_i^+ - d_i^-$$

where:

- d_i^+ is the positive degree of node i
- d_i^- is the negative degree of node i

Python Implementation: ` ` `python def signed_degree_centrality(G): """Calculate signed degree centrality""" centrality = {}

```
for node in G.nodes():
    pos_degree = sum(1 for _, _, data in G.edges(node, data=True)
                    if data.get('sign', 1) == 1)
    neg_degree = sum(1 for _, _, data in G.edges(node, data=True)
                    if data.get('sign', 1) == -1)

    centrality[node] = {
        'positive_degree': pos_degree,
        'negative_degree': neg_degree,
        'net_degree': pos_degree - neg_degree
    }

return centrality
```

` ` `

Interpretation:

- High positive net degree: Well-liked, influential
- High negative net degree: Controversial, many enemies
- Near zero: Balanced relationships

---

2. Walk-Based Centrality (RECOMMENDED PRIMARY METHOD)

> ✅ ***RECOMMENDED***: *Walk-based centrality is the most reliable method for signed networks. It properly accounts for how influence propagates along all paths with signs multiplying.*

Reference: Liu et al. (2020) "A simple approach for quantifying node centrality in signed and directed social networks"

Key Innovation: Instead of shortest paths, consider ALL walks where effects propagate and signs multiply.

Direct Effect Formula

The direct effect of node i on node j is:

$$a_{ij} = \frac{\sigma_{ij}}{d_j}$$

where:

- σ_{ij} is the sign of edge (i, j) ($+1$ or -1)
- d_j is the degree of node j

Intuition: Effect is stronger when target has fewer connections.

Indirect Effects and Sign Propagation

For a walk $i \to k \to j$, the indirect effect is:

$$\text{effect}(i \to j \text{ via } k) = a_{ik} \times a_{kj}$$

Sign Propagation Rules:

- $(+) \times (+) = +$: Positive influence through positive intermediary
- $(+) \times (-) = -$: Positive becomes negative through negative intermediary
- $(-) \times (-) = +$: Negative through negative (enemy of enemy)

Total Effect (TE)

Definition: Sum of effects along ALL walks up to length n:

$$TE_{ij}^{(n)} = \sum_{l=1}^{n} \sum_{\text{walks of length } l} \text{effect along walk}$$

Total Effect of node i on whole network:

$$TE_i = \sum_{j=1}^{N} TE_{ij}^{(n)}$$

Interpretation: Overall influence magnitude, regardless of sign.

Net Effect (NE)

Definition: Positive effects minus negative effects:

$$NE_{ij}^{(n)} = E_{ij}^{(n)+} - E_{ij}^{(n)-}$$

Net Effect of node i:

$$NE_i = \sum_{j=1}^{N} NE_{ij}^{(n)}$$

Interpretation:

- $NE_i > 0$: Predominantly positive influence
- $NE_i < 0$: Predominantly negative influence
- $NE_i \approx 0$: Balanced positive and negative influence

Python Implementation

```
def signed_walk_effect(G, max_steps=3):
    """
    Calculate walk-based centrality for signed networks.

    Returns total effect and net effect for each node.
    This is the RECOMMENDED method for signed networks.
    """
```

```
    nodes = list(G.nodes())
    n = len(nodes)
    node_to_idx = {node: i for i, node in enumerate(nodes)}

    # Initialize direct effect matrix
    A = np.zeros((n, n))
    for u, v, data in G.edges(data=True):
        i, j = node_to_idx[u], node_to_idx[v]
        sign = data.get('sign', 1)
        degree_v = G.degree(v)
        if degree_v > 0:
            A[i, j] = sign / degree_v
            # For undirected graphs
            A[j, i] = sign / degree_v

    # Calculate cumulative effects up to max_steps
    total_effect = np.eye(n)  # Start with self-effect
    current = A.copy()

    for step in range(1, max_steps + 1):
        total_effect += current
        current = current @ A  # Matrix multiplication for next
step

    # Calculate metrics for each node
    results = {}
    for i, node in enumerate(nodes):
        row_sum = np.sum(total_effect[i, :])  # Total effect
exerted

        # Separate positive and negative effects
        positive_effect = np.sum(total_effect[i, :]
[total_effect[i, :] > 0])
        negative_effect = np.sum(np.abs(total_effect[i, :]
[total_effect[i, :] < 0]))
        net_effect = positive_effect - negative_effect

        results[node] = {
            'total_effect': row_sum,
            'net_effect': net_effect,
            'positive_effect': positive_effect,
            'negative_effect': negative_effect
```

```
        }

    return results
```

Usage Example: ```python

Calculate walk-based centrality (RECOMMENDED)

centrality = signed_walk_effect(G, max_steps=3)

Find most influential nodes

most_influential = max(centrality.items(), key=lambda x: x[1]['total_effect']) most_positive = max(centrality.items(), key=lambda x: x[1]['net_effect'])

print(f"Most influential: {most_influential[0]}") print(f"Most positive influence: {most_positive[0]}") ```

---

3. Signed Betweenness Centrality (Approximation Only)

> ⚠️ ***WARNING**: No consensus on "correct" signed betweenness in literature. Use approximations with caution or prefer walk-based methods.*

Challenge: Standard betweenness assumes all paths are equally "good" for communication.

Approach: Structure-Weighted Approximation

```
def signed_betweenness_approximation(G):
```

```
    """
    Approximate betweenness for signed networks.

    WARNING: This is an approximation with limitations.
    Consider using walk-based centrality instead.
    """
    # Create unsigned version for path counting
    G_unsigned = nx.Graph()
    G_unsigned.add_nodes_from(G.nodes())
    G_unsigned.add_edges_from(G.edges())

    # Calculate standard betweenness
    betweenness = nx.betweenness_centrality(G_unsigned)

    # Weight by local sign environment
    weighted_betweenness = {}
    for node in G.nodes():
        pos_edges = sum(1 for _, _, d in G.edges(node, data=True)
                       if d.get('sign', 1) == 1)
        neg_edges = sum(1 for _, _, d in G.edges(node, data=True)
                       if d.get('sign', 1) == -1)
        total_edges = pos_edges + neg_edges

        if total_edges > 0:
            sign_ratio = (pos_edges - neg_edges) / total_edges
            weighted_betweenness[node] = betweenness[node] * (1 +
sign_ratio) / 2
        else:
            weighted_betweenness[node] = 0.0

    return weighted_betweenness
```

---

4. Signed Closeness Centrality

Challenge: What is "distance" through negative edges?

Solution: Harmonic Closeness with Positive Edges Only

```
def signed_closeness_harmonic(G):
    """
    Harmonic closeness using only positive edges.

    Only considers reachability through friendly relationships.
    """
    # Create subgraph with only positive edges
    G_positive = nx.Graph()
    G_positive.add_nodes_from(G.nodes())
    for u, v, data in G.edges(data=True):
        if data.get('sign', 1) == 1:
            G_positive.add_edge(u, v)

    closeness = {}
    for node in G.nodes():
        harmonic_sum = 0.0
        for target in G.nodes():
            if node != target:
                try:
                    distance = nx.shortest_path_length(G_positive,
node, target)
                    harmonic_sum += 1.0 / distance
                except nx.NetworkXNoPath:
                    pass  # No path through positive edges

        n = len(G.nodes())
        closeness[node] = harmonic_sum / (n - 1) if n > 1 else 0

    return closeness
```

Interpretation:

- High closeness: Well-connected through friendly relationships
- Low closeness: Isolated or only reachable through enemies

---

5. Signed Eigenvector Centrality

🚨 ***CRITICAL WARNING***: *Eigenvector centrality* ***may not exist*** *for signed networks! The Perron-Frobenius theorem does NOT apply when adjacency matrix has negative entries.*

When Eigenvector Centrality Fails:

1. No dominant positive eigenvalue
2. Multiple eigenvalues with same magnitude
3. Complex eigenvalues with imaginary components
4. Eigenvector with mixed signs (no clear interpretation)

Mathematical Definition:

$$x_i = \frac{1}{\lambda} \sum_j A_{ij} x_j$$

where A_{ij} can be positive or negative.

Python Implementation with Safety Checks:

```
def signed_eigenvector_centrality(G, tol=1e-6):
    """
    Calculate eigenvector centrality for signed networks.

    WARNING: May not exist! Always check return status.

    Returns
    -------
    centrality : dict or None
        Centrality values if computable, None otherwise
    status : str
        Status message explaining result
    """
    nodes = list(G.nodes())
    n = len(nodes)
```

```
    if n == 0:
        return None, "Empty graph"

    # Build signed adjacency matrix
    A = nx.adjacency_matrix(G, nodelist=nodes,
weight='sign').toarray()

    # Find eigenvalues
    try:
        eigenvalues, eigenvectors = np.linalg.eig(A)
        eigenvalues = eigenvalues.real
    except:
        return None, "Failed to compute eigenvalues"

    # Find largest eigenvalue by magnitude
    max_idx = np.argmax(np.abs(eigenvalues))
    lambda_max = eigenvalues[max_idx]

    # Check 1: Dominant eigenvalue?
    sorted_eigs = sorted(np.abs(eigenvalues), reverse=True)
    if len(sorted_eigs) > 1 and np.abs(sorted_eigs[0] -
sorted_eigs[1]) < tol:
        return None, "No dominant eigenvalue (multiple with same
magnitude)"

    # Check 2: Positive eigenvalue?
    if lambda_max <= 0:
        return None, f"No positive dominant eigenvalue (λ=
{lambda_max:.4f})"

    # Get eigenvector
    eigenvector = eigenvectors[:, max_idx].real

    # Check 3: All same sign?
    if np.all(eigenvector >= 0) or np.all(eigenvector <= 0):
        eigenvector = np.abs(eigenvector)
        eigenvector = eigenvector / eigenvector.sum()
        centrality = {node: eigenvector[i] for i, node in
enumerate(nodes)}
        return centrality, f"Success (λ={lambda_max:.4f})"
    else:
        return None, "Eigenvector has mixed signs"
```

```
# Usage
centrality, status = signed_eigenvector_centrality(G)
if centrality is None:
    print(f"Cannot compute: {status}")
    print("Using walk-based centrality instead")
    centrality = signed_walk_effect(G)
```

Reference: Bonacich & Lloyd (2004) "Calculating status with negative relations"

Comparison: Which Centrality to Use?

Measure	Best For	Reliability	Recommendation
Walk-Based (TE/NE)	Overall influence, signed impact	✅✅✅ Always works	**PRIMARY METHOD**
Signed Degree	Quick assessment	✅✅✅ Always works	Good supplement
Signed Eigenvector	Friend-of-friend influence	⚠️ May not exist	Check first, use cautiously
Signed Betweenness	Bridge identification	⚠️ Approximation only	Use with caution
Signed Closeness	Reachability	⚠️ Multiple definitions	Use specific interpretation

Primary Recommendation: Use **walk-based centrality** (total effect and net effect) as your primary measure, supplemented with signed degree for quick insights.

Community Detection

> ⚠️ ***CRITICAL WARNING**: Standard `nx.algorithms.community.modularity()` does NOT work for signed networks! It completely ignores edge signs.*

1. Signed Modularity

The Problem: Standard Newman modularity treats all edges equally.

Correct Formula: Gómez et al. signed modularity with parameter α:

$$Q_{signed} = \alpha \cdot Q(G^+) + (1 - \alpha) \cdot Q(G^-)$$

Where:

- $Q(G^+)$: Modularity for positive edges (want within communities)
- $Q(G^-)$: Modularity for negative edges (want between communities)
- $\alpha \in [0, 1]$: Balance parameter (typically 0.5)

Python Implementation:

```
def signed_modularity(G, communities, alpha=0.5):
    """
    Calculate signed modularity correctly.

    Parameters
    ----------
    G : networkx.Graph
        Signed network with 'sign' edge attribute
    communities : dict
        Node to community assignment
    alpha : float (0 to 1)
        Balance between positive (1.0) and negative (0.0) edges
        Default 0.5 = equal weight

    Returns
    -------
    float
        Signed modularity value

    References
```

```
----------
Gómez et al. (2009), Traag & Bruggeman (2009)
"""
# Separate into positive and negative subgraphs
G_pos = nx.Graph()
G_neg = nx.Graph()
G_pos.add_nodes_from(G.nodes())
G_neg.add_nodes_from(G.nodes())

m_pos = 0
m_neg = 0

for u, v, data in G.edges(data=True):
    sign = data.get('sign', 1)
    if sign > 0:
        G_pos.add_edge(u, v)
        m_pos += 1
    else:
        G_neg.add_edge(u, v)
        m_neg += 1

# Calculate Q for positive edges (want within communities)
Q_pos = 0.0
if m_pos > 0:
    for comm_id in set(communities.values()):
        nodes_in_comm = [n for n, c in communities.items()
                       if c == comm_id]
        subgraph = G_pos.subgraph(nodes_in_comm)
        l_c = subgraph.number_of_edges()
        d_c = sum(G_pos.degree(n) for n in nodes_in_comm)
        Q_pos += (l_c / m_pos) - (d_c / (2 * m_pos)) ** 2

# Calculate Q for negative edges (want between communities)
Q_neg = 0.0
if m_neg > 0:
    between_edges = sum(1 for u, v in G_neg.edges()
                      if communities[u] != communities[v])
    Q_neg = between_edges / m_neg

return alpha * Q_pos + (1 - alpha) * Q_neg
```

⚠️ **Known Issue**: As the number of negative ties increases, the density of positive ties is neglected more (Esmailian & Jalili, 2015).

2. Signed Spectral Clustering

Algorithm: ```python def signed_spectral_clustering(G, num_communities=2): """Spectral clustering for signed networks""" nodes = list(G.nodes()) n = len(nodes)

```
# Create signed adjacency matrix
A = nx.adjacency_matrix(G, weight='sign').toarray()

# Compute signed Laplacian
D = np.diag(np.sum(np.abs(A), axis=1))
L = D - A

# Find eigenvectors
eigenvalues, eigenvectors = np.linalg.eigh(L)

# Use Fiedler vector (second smallest eigenvalue)
fiedler_vector = eigenvectors[:, 1]

# Partition based on sign
communities = {}
for i, node in enumerate(nodes):
    communities[node] = 0 if fiedler_vector[i] < 0 else 1

return communities
```

```

### 3. Community Quality Metrics

```
def analyze_community_quality(G, communities):
    """Analyze quality of detected communities in signed
network"""
    internal_pos = 0 # Good
    internal_neg = 0 # Bad
    external_pos = 0 # Bad
    external_neg = 0 # Good
```
```

```
    for u, v, data in G.edges(data=True):
        sign = data['sign']
        same_community = (communities[u] == communities[v])

        if same_community:
            if sign == 1:
                internal_pos += 1
            else:
                internal_neg += 1
        else:
            if sign == 1:
                external_pos += 1
            else:
                external_neg += 1

    good_edges = internal_pos + external_neg
    bad_edges = internal_neg + external_pos
    quality = good_edges / (good_edges + bad_edges) if (good_edges
+ bad_edges) > 0 else 0

    return {
        'internal_positive': internal_pos,
        'internal_negative': internal_neg,
        'external_positive': external_pos,
        'external_negative': external_neg,
        'quality_score': quality
    }
```

Link Prediction

Balance-Based Prediction

Principle: Predict signs that maximize structural balance.

Algorithm:

```
def predict_edge_sign(G, node1, node2):
```

```
    """
    Predict edge sign using balance theory.

    Returns
    -------
    predicted_sign : int (1 or -1)
        Predicted sign
    confidence : float
        Prediction confidence
    """
    # Find common neighbors
    neighbors1 = set(G.neighbors(node1))
    neighbors2 = set(G.neighbors(node2))
    common_neighbors = neighbors1.intersection(neighbors2)

    if len(common_neighbors) == 0:
        return None, 0.0

    vote_positive = 0
    vote_negative = 0

    for neighbor in common_neighbors:
        sign1 = G[node1][neighbor]['sign']
        sign2 = G[node2][neighbor]['sign']

        # Balance rule: if signs match → predict positive
        if sign1 * sign2 == 1:
            vote_positive += 1
        else:
            vote_negative += 1

    predicted_sign = 1 if vote_positive > vote_negative else -1
    confidence = max(vote_positive, vote_negative) /
(vote_positive + vote_negative)

    return predicted_sign, confidence
```

Example: ` ` ` A ---?--- B \ / (+)\ /(+) \ / C

Prediction: A-B should be POSITIVE Reasoning: (+)(+)(+) = + (balanced triangle) Confidence: 100% (1 common neighbor, 1 vote) ` ` `

Applications

1. Social Networks

Online Social Networks

- **Positive edges**: Friends, followers, likes
- **Negative edges**: Blocks, unfriends, dislikes
- **Applications**: Recommendation systems, sentiment analysis

Political Networks

- **Positive edges**: Alliances, agreements
- **Negative edges**: Conflicts, disagreements
- **Applications**: Political analysis, conflict resolution

2. Economic Networks

Trade Networks

- **Positive edges**: Trade agreements, partnerships
- **Negative edges**: Trade disputes, sanctions
- **Applications**: Economic modeling, policy analysis

3. Biological Networks

Protein Interaction Networks

- **Positive edges**: Activating interactions
- **Negative edges**: Inhibiting interactions
- **Applications**: Drug discovery, disease understanding

Gene Regulatory Networks

- **Positive edges**: Gene activation
- **Negative edges**: Gene repression
- **Applications**: Gene therapy, disease treatment

Complete Analysis Example

```
import networkx as nx
import numpy as np

# Create signed network
G = nx.Graph()
G.add_edges_from([
    ('A', 'B', {'sign': 1}),
    ('B', 'C', {'sign': 1}),
    ('C', 'A', {'sign': 1}),  # Community 1: all friends
    ('D', 'E', {'sign': 1}),
    ('E', 'F', {'sign': 1}),  # Community 2: friends
    ('A', 'D', {'sign': -1}),
    ('B', 'E', {'sign': -1}),
    ('C', 'F', {'sign': -1})  # Between communities: enemies
])

# 1. Calculate balance index
balance_idx = calculate_balance_index(G)
print(f"Balance Index: {balance_idx:.3f}")

# 2. Calculate centrality (USE WALK-BASED - RECOMMENDED)
centrality = signed_walk_effect(G, max_steps=3)
print("\nTop 3 by Total Effect:")
sorted_nodes = sorted(centrality.items(),
                     key=lambda x: x[1]['total_effect'],
                     reverse=True)[:3]
for node, metrics in sorted_nodes:
    print(f"  {node}: TE={metrics['total_effect']:.3f}, "
          f"NE={metrics['net_effect']:.3f}")

# 3. Detect communities
communities = signed_spectral_clustering(G, num_communities=2)
print(f"\nCommunities: {communities}")

# 4. Calculate CORRECT signed modularity
Q_signed = signed_modularity(G, communities, alpha=0.5)
print(f"Signed Modularity: {Q_signed:.3f}")

# 5. Evaluate community quality
```

```
quality = analyze_community_quality(G, communities)
print(f"Community Quality: {quality['quality_score']:.3f}")

# 6. Link prediction
for node1 in ['A', 'D']:
    for node2 in ['B', 'E']:
        if not G.has_edge(node1, node2):
            pred_sign, conf = predict_edge_sign(G, node1, node2)
            if pred_sign:
                sign_str = '+' if pred_sign == 1 else '-'
                print(f"Predict {node1}-{node2}: {sign_str} "
                      f"(confidence: {conf:.2%})")
```

Key Takeaways

1. ❌ **Standard centrality measures DON'T work** - they ignore edge signs completely
2. ✅ **Use walk-based centrality** (total effect & net effect) as primary method
3. ❌ **Standard modularity FAILS** - use signed modularity with α parameter
4. ⚠️ **Eigenvector centrality may NOT EXIST** - always check before computing
5. **Balance theory predicts** stable network configurations
6. **Always cite proper papers** - don't cite unsigned network methods

---

References

Critical Papers for Signed Networks

1. **Harary, F. (1953)**. "On the notion of balance of a signed graph." Michigan Mathematical Journal, 2(2), 143-146.
2. Original balance theory
3. **Cartwright, D., & Harary, F. (1956)**. "Structural balance: a generalization of Heider's theory." Psychological Review, 63(5), 277.

4. Extended balance theory
5. **Liu, W. C., Huang, L. C., Liu, C. W. J., & Jordán, F. (2020)**. "A simple approach for quantifying node centrality in signed and directed social networks." Applied Network Science, 5(1), 1-18.
6. **Walk-based centrality (RECOMMENDED METHOD)**
7. Read online
8. **Bonacich, P., & Lloyd, P. (2004)**. "Calculating status with negative relations." Social Networks, 26(4), 331-338.
9. Eigenvector centrality for signed networks
10. **Traag, V. A., & Bruggeman, J. (2009)**. "Community detection in networks with positive and negative links." Physical Review E, 80(3), 036115.
11. Signed modularity
12. **Esmailian, P., & Jalili, M. (2015)**. "Community detection in signed networks: the role of negative ties in different scales." Scientific Reports, 5, 14339.
13. Shows inconsistencies in signed modularity
14. **Everett, M. G., & Borgatti, S. P. (2014)**. "Networks containing negative ties." Social Networks, 38, 111-120.
15. PN centrality measure
16. **Leskovec, J., Huttenlocher, D., & Kleinberg, J. (2010)**. "Predicting positive and negative links in online social networks." Proceedings of the 19th International Conference on World Wide Web, 641-650.
17. Link prediction in signed networks