NETWORK SCIENCE PROJECT

UNVEILING THE BITCOIN ALPHA TRUST NETWORK - A NETWORK SCIENCE APPROACH

DELIVERABLE 3

GROUP 33

VAIBHAV GUPTA - 2022553
RATNANGO GHOSH - 2022397

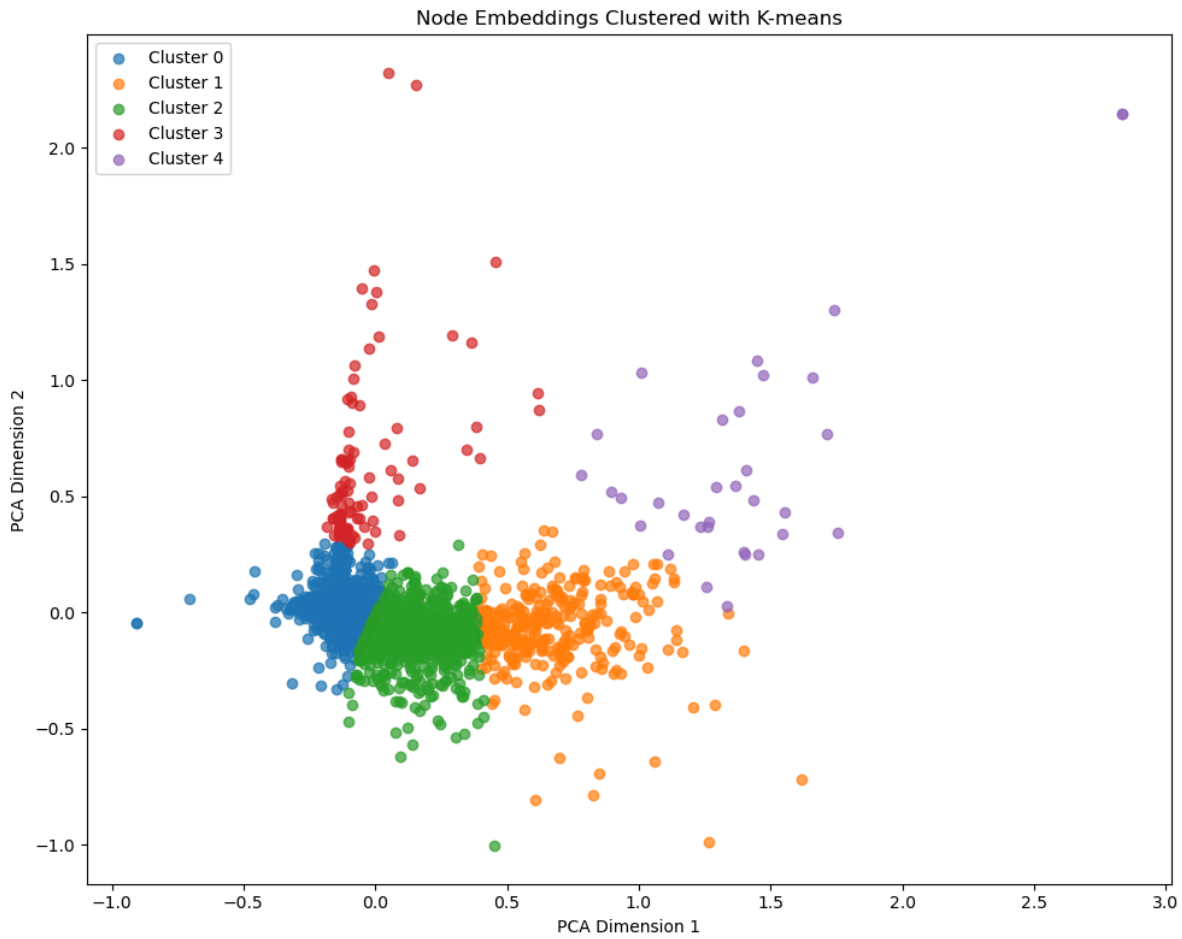
1. Dataset Overview

The Bitcoin Alpha dataset contains 24,186 trust ratings between users, representing a significant sample of platform interactions:

- **Ratings Range**: -10 (strong distrust) to +10 (strong trust)
- **Network Size**: 3,783 unique users
- **Unique Sources**: 3,286 users providing ratings
- **Unique Targets**: 3,754 users receiving ratings
- **Rating Distribution**:
    - Positive ratings: 22,650 (93.6%)
    - Negative ratings: 1,536 (6.4%)

This overwhelming positive bias (93.6%) suggests a general trend toward trust within the community, but also raises questions about potential rating inflation or social pressure to provide positive ratings.

2. Anomalous Trust Clusters



Our analysis identified two particularly anomalous clusters that exhibit unusual internal trust patterns compared to their external relationships:

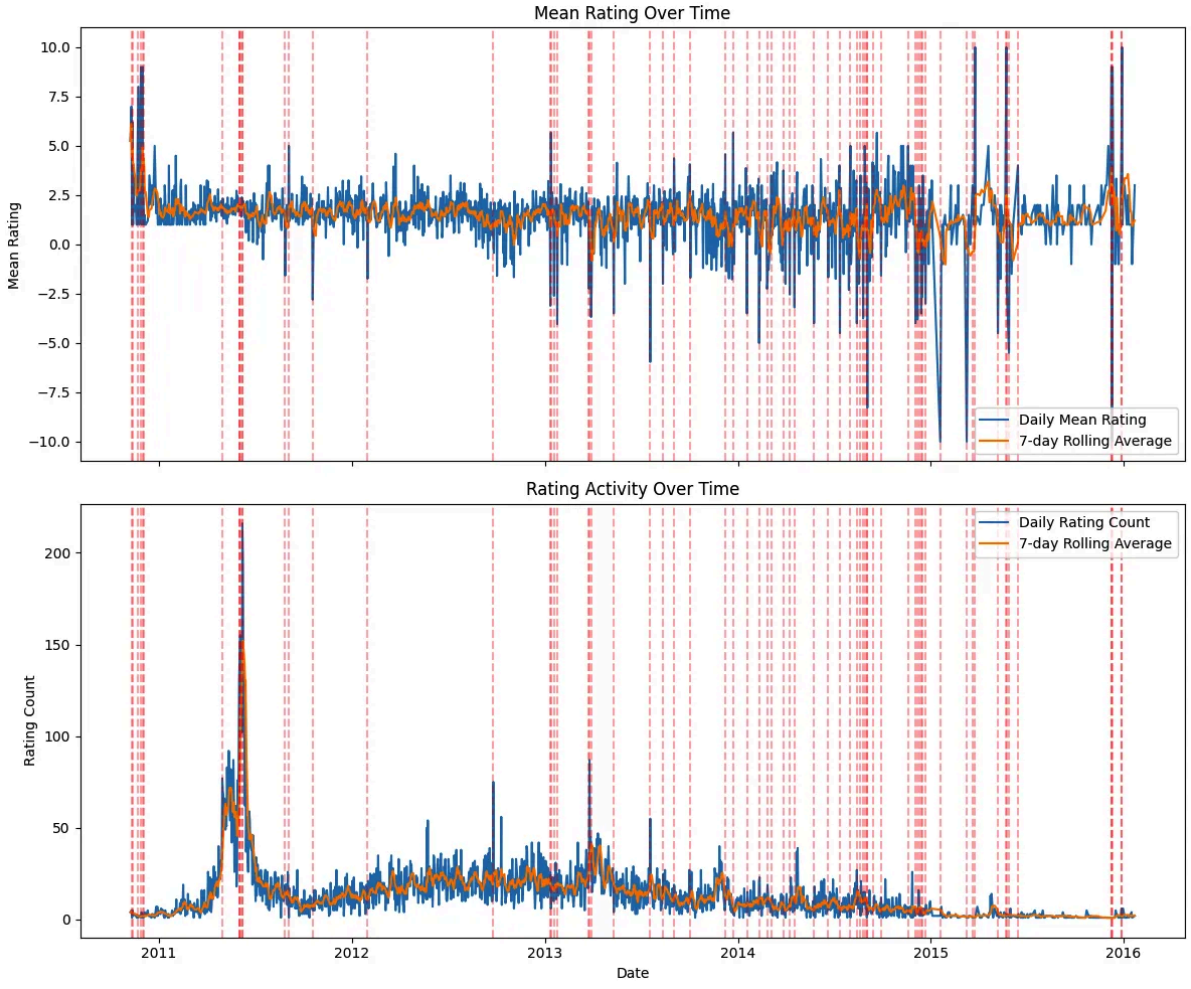
Cluster ID	Size	Internal Trust Mean	External Trust Mean	Trust Ratio
1	248	3.56	1.75	2.03
4	76	6.23	3.18	1.96

Cluster 1 (248 users) shows internal trust levels that are approximately twice as high as external trust levels, suggesting a potential "echo chamber" effect or coordinated group. Cluster 4 (76 users) displays even higher internal trust values with a similar ratio to external trust.

These clusters may represent:

- Community subgroups with stronger internal cohesion
- Trading circles with established trust relationships
- Potential manipulation rings artificially boosting internal trust scores

3. Temporal Anomalies

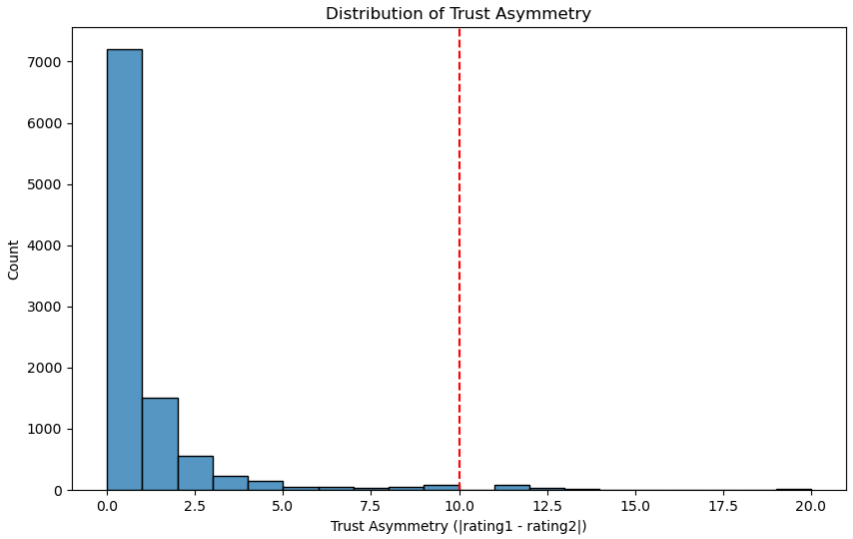


The temporal analysis revealed 71 days with statistically unusual rating patterns between 2010-2015. Key observations from the time series analysis:

- **Early Platform Activity (2011)**: Shows high variance with extreme rating values, possibly due to early platform adoption dynamics
- **Peak Activity Period (2011-2012)**: Shows highest rating volumes and relatively stable mean ratings
- **Late 2010 Anomalies**: Notable extreme trust ratings in late November 2010
- **December 2015 Extreme Ratings**: Particularly anomalous day (2015-12-10) with a -10 rating average

The most active users during anomalous periods include users with IDs 7564 (45 ratings), 130 (20 ratings), and 7604 (20 ratings). This concentration of activity suggests potential coordinated behavior or unusual influence by specific users during volatile periods.

4. Trust Asymmetry Patterns



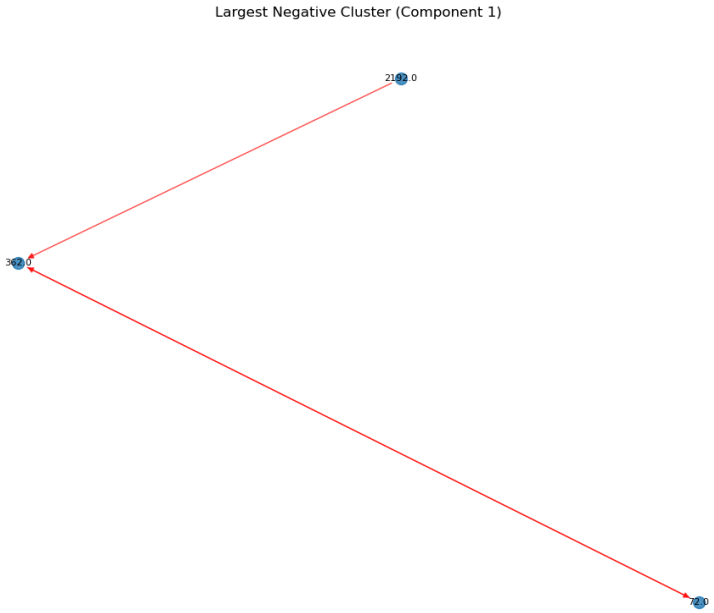
The analysis identified 248 highly asymmetric relationships among 10,062 bidirectional connections. The most extreme cases show maximum possible asymmetry (20 points):

User Pair	Rating Direction 1	Rating Direction 2	Asymmetry
5 → 11	-10	+10	20
141 → 7481	-10	+10	20
7 → 142	-10	+10	20
157 → 7604	-10	+10	20
838 → 7335	-10	+10	20

These extreme asymmetries represent fundamental disagreements about trustworthiness or potential retaliatory behavior between users. The pattern of maximum negative rating in one direction and maximum positive in the other suggests possible:

- Retaliatory ratings after business disputes
- Attempted manipulation of competitor reputations
- Fundamental disagreements about trustworthiness within user pairs

5. Negative Trust Subgraphs



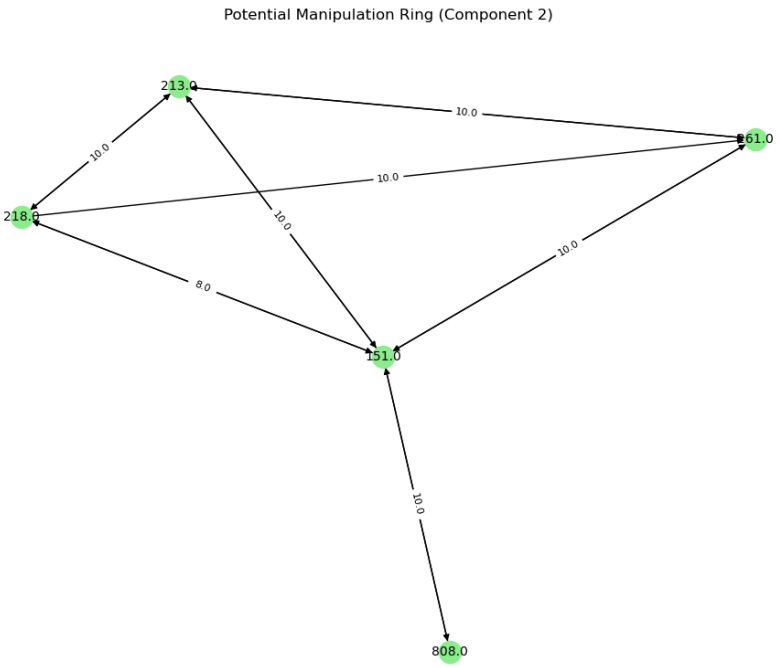
The analysis identified 7 connected components in the negative trust network with 3+ nodes. Four particularly notable negative clusters were detected:

Component ID	Size	Positive Edges	Negative Edges	Negative Ratio
1	3	0	3	1.00
2	3	0	2	1.00
4	4	0	4	1.00
6	4	1	3	0.75

Components 1, 2, and 4 are particularly interesting as they contain exclusively negative connections, suggesting strong mutual distrust or adversarial relationships. These subgraphs may represent:

- Competing trade groups
- Users involved in disputed transactions
- Users identifying suspected scammers

6. Manipulation Rings



Five potential manipulation rings were identified through analysis of strong positive trust components:

Component ID	Size	Density	Average Rating	Reciprocity	Rating Difference
1	3	1.00	10.00	1.00	8.80
2	5	0.65	9.85	0.60	8.55
4	3	1.00	9.17	0.67	6.71
14	4	0.58	10.00	0.50	8.65
16	4	0.58	10.00	0.33	8.50

The most suspicious ring (Component 1) shows:

- Complete connectivity (density 1.0)
- Maximum possible trust ratings (10.0)
- Perfect reciprocity (1.0)
- Dramatic difference (8.8) between internal and external ratings

These characteristics strongly suggest coordinated artificial inflation of trust scores within a small group of users. Components 2, 14, and 16 also show extremely high ratings (≈10) with significant differences from external ratings, indicating possible manipulation.

7. Rating Activity Patterns

- **Early 2011 Spike**: A dramatic increase in platform activity, with daily counts exceeding 200 ratings
- **General Decline**: A gradual decrease in platform activity from 2012-2015
- **Anomalous Activity Periods**: Several days with activity spikes throughout the network's history (indicated by red dashed lines)
- **Low Recent Activity**: By 2015-2016, daily rating activity had significantly decreased

This pattern suggests a network that experienced early rapid growth, followed by stabilization and gradual decline, with occasional bursts of unusual activity.

8. Code Explanation

8.1 Environment Setup and Imports

GPU-enabled PyTorch and PyG installs (with CPU fallbacks commented out)

```
!pip install torch==2.0.0+cu118 torchvision==0.15.1+cu118 torchaudio==2.0.1
--extra-index-url https://download.pytorch.org/whl/cu118

!pip install -q torch-scatter torch-sparse torch-cluster torch-spline-conv -f
https://data.pyg.org/whl/torch-2.0.0+cu118.html

!pip install -q torch-geometric
!pip install numpy==1.24.0
import pandas as pd
import numpy as np
import networkx as nx
import torch
import torch.nn.functional as F
from torch_geometric.nn import GCNConv
from torch_geometric.data import Data
from sklearn.cluster import KMeans, DBSCAN
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
import seaborn as sns
from collections import defaultdict, Counter
import datetime
import matplotlib.dates as mdates
```

PyTorch & PyG: Installs GPU-compatible wheels (with CPU alternatives).
pandas/numpy: Data handling and numeric ops.
networkx: Classic graph building and analysis.
torch_geometric: Graph Neural Network layers (`GCNConv`) and data structures (`Data`).
scikit-learn: Clustering (`KMeans`, `DBSCAN`), PCA, feature scaling.
matplotlib/seaborn: Visualization of temporal and structural anomalies.

8.2 Class Definition and Attributes

```
class AnomalousTrustDetector:
    def __init__(self, filepath):
        self.filepath      = filepath      CSV path
        self.df            = None          Raw DataFrame
        self.nx_G          = None          NetworkX DiGraph
        self.edge_index    = None          PyG adjacency
        self.edge_attr     = None          Edge weights tensor
        self.edge_time     = None          Edge timestamps tensor
        self.node_features = None          Precomputed node feature tensor
        self.gnn_model     = None          Trained GNN
        self.node_embeddings = None        Learned node embeddings
```

State holders for raw data, graphs, features, model, and results.

8.3 Loading & Preprocessing Data

```
def load_data(self):
     1. Read CSV with columns [source, target, rating, time]
    self.df = pd.read_csv(self.filepath, names=['source','target','rating','time'])
     2. Print summary stats: counts, unique users, rating distribution
     3. Convert UNIX timestamps to datetime for temporal analysis
    self.df['datetime'] = pd.to_datetime(self.df['time'], unit='s')
     4. Build graphs
    return self.create_graph()
```

Robust loading: catches exceptions and attempts alternate parsing.

Summary: dataset size, nodes, rating range, positive vs negative percentages.

Datetime: needed downstream for rolling-window and anomaly detection.

8.4 Graph Construction

```
def create_graph(self):
     Build a directed NetworkX graph with weight=rating and time attributes
    self.nx_G = nx.DiGraph()
    for _, row in self.df.iterrows():
        self.nx_G.add_edge(row['source'], row['target'], weight=row['rating'], time=row['time'])
```

```
    Map node IDs to contiguous indices for PyG
    all_nodes = sorted(self.nx_G.nodes())
    node_idx = {node: i for i, node in enumerate(all_nodes)}

    Build edge_index and edge_attr tensors for torch_geometric
    edge_list, weight_list, time_list = [], [], []
    for u, v, data in self.nx_G.edges(data=True):
        edge_list.append([node_idx[u], node_idx[v]])
        weight_list.append(data['weight'])
        time_list.append(data['time'])

    self.edge_index = torch.tensor(edge_list, dtype=torch.long).t().contiguous()
    self.edge_attr  = torch.tensor(weight_list, dtype=torch.float).view(-1, 1)
    self.edge_time  = torch.tensor(time_list, dtype=torch.long)
    self.num_nodes  = len(all_nodes)

    Compute node features and assemble PyG Data
    self.compute_node_features(node_idx)
    self.data = Data(x=self.node_features,
                     edge_index=self.edge_index,
                     edge_attr=self.edge_attr,
                     num_nodes=self.num_nodes)
    return True
```

NetworkX→PyG: translates from `.edges()` to `edge_index` (2×E) and `edge_attr`.

Index mapping: ensures nodes are 0…N–1.

8.5 Node Feature Engineering

```
def compute_node_features(self, node_idx):
    Initialize a (N×6) array:
    [in_degree, out_degree, avg_in_rating, avg_out_rating, var_in_rating, pct_negative_in]
    features = np.zeros((self.num_nodes, 6))
    for node, idx in node_idx.items():
        in_edges  = list(self.nx_G.in_edges(node, data=True))
        out_edges = list(self.nx_G.out_edges(node, data=True))
        in_ratings  = [d['weight'] for _, d in in_edges] or [0]
        out_ratings = [d['weight'] for _, d in out_edges] or [0]

        features[idx] = [
            len(in_edges),
            len(out_edges),
            np.mean(in_ratings),
```

```
            np.mean(out_ratings),
            np.var(in_ratings),
            sum(r<0 for r in in_ratings)/len(in_ratings)
        ]
    Standardize features to zero mean/unit variance
    scaler = StandardScaler()
    self.node_features = torch.tensor(scaler.fit_transform(features), dtype=torch.float)
```

Captures basic trust behavior per node: activity, sentiment, variability.

Scaling ensures features contribute comparably in the GNN.

8.6 Building & Training the GNN

```
def build_gnn_model(self, hidden_channels=64):
    Define a 3-layer GCN
    class GNN(torch.nn.Module):
        def __init__(self, in_c, hid_c, out_c):
            super().__init__()
            self.conv1 = GCNConv(in_c, hid_c)
            self.conv2 = GCNConv(hid_c, hid_c)
            self.conv3 = GCNConv(hid_c, out_c)
        def forward(self, x, edge_index, edge_weight=None):
            x = F.relu(self.conv1(x, edge_index, edge_weight))
            x = F.dropout(x, p=0.2, training=self.training)
            x = F.relu(self.conv2(x, edge_index, edge_weight))
            return self.conv3(x, edge_index, edge_weight)

    Instantiate, optimizer, and simple self-supervised loop
    self.gnn_model = GNN(self.node_features.size(1), hidden_channels, out_channels=32)
    optimizer = torch.optim.Adam(self.gnn_model.parameters(), lr=0.01)
    edge_weight = torch.ones_like(self.edge_attr.view(-1))

    self.gnn_model.train()
    for epoch in range(100):
        optimizer.zero_grad()
        embeddings = self.gnn_model(self.node_features, self.edge_index, edge_weight)
         Reconstruct edge weights via dot-product of source/dest embeddings
        src, dst = self.edge_index
        pred = (embeddings[src]  embeddings[dst]).sum(dim=1)
        target = (self.edge_attr.view(-1) - self.edge_attr.mean()) / (self.edge_attr.std() + 1e-8)
        loss = F.mse_loss(pred, target)
        loss.backward(); optimizer.step()
        if (epoch+1) % 20 == 0:
            print(f'Epoch {epoch+1}, Loss {loss:.4f}')

    self.gnn_model.eval()
    with torch.no_grad():
```

self.node_embeddings = self.gnn_model(self.node_features, self.edge_index, edge_weight).cpu().numpy()

Architecture: three stacked GCNConv layers with ReLU + dropout.

Objective: unsupervised “reconstruct” edge weights via embedding dot-products.

Output: 32-dim vector per node for downstream anomaly searches.

8.7 Anomaly Detection Methods

8.7.1 Anomalous Clusters

```
def detect_anomalous_clusters(self, n_clusters=5):
    clusters = KMeans(n_clusters, random_state=42).fit_predict(self.node_embeddings)
     For each cluster, compute:
      – mean “inner” vs “outer” trust
      – trust_ratio = inner_mean / outer_mean
     Flag clusters if:
       trust_ratio > 1.5, inner_mean >5 and >2×outer_mean, or low inner_std + size>10
     Visualize via PCA scatter and return list of suspicious cluster IDs.
```

Spots groups whose internal trust diverges sharply from outside interactions.

8.7.2 Temporal Anomalies

```
def detect_temporal_anomalies(self):
     Aggregate daily mean rating and count
    daily = self.df.set_index('datetime').resample('D')['rating'].agg(['mean','count'])
     Compute 7-day rolling means and z-scores
    daily['mean_rolling']  = daily['mean'].rolling(7).mean()
    daily['count_rolling'] = daily['count'].rolling(7).mean()
    daily['mean_z']   = (daily['mean']-daily['mean'].rolling(7).mean()) / daily['mean'].std()
    daily['count_z']  = (daily['count']-daily['count'].rolling(7).mean()) / daily['count'].std()
     Days with |mean_z|>2 or count_z>2 are anomalous
     Plot time series with red lines on outliers, list most active users on those days.
```

Captures spikes/drops in daily volume or average rating.

8.7.3 Trust Asymmetry

```
def detect_trust_asymmetry(self):
     For every bidirectional pair (u,v), record both ratings
     Compute asymmetry = |rating(u→v) – rating(v→u)|
     Flag pairs with asymmetry >10 or opposite signs
     Plot histogram of asymmetries, return top offenders.
```

Highlights pairs with sharp disagreements or retaliatory extremes.

8.7.4 Negative Subgraphs

```
def detect_negative_subgraphs(self):
    Build subgraph of only negative edges
   neg_G = nx.DiGraph([(u,v) for u,v,d in self.nx_G.edges(data=True) if d['weight']<0])
    Find undirected connected components of size≥3
    For each, compute negative_ratio = neg_edges/total_edges
    Flag components with negative_ratio>0.5, visualize the largest.
```

Identifies clusters dominated by distrust.

8.7.5 Sybil Attack Detection

```
def detect_sybil_attacks(self):
    Bin users by “first appearance” timestamp into 20 windows
    Find windows with unusually high new-user counts
    For each window, compute:
      – internal vs external positive rating ratios
      – connectivity and time span
    Flag windows where a tight-knit, highly positive group forms rapidly.
```

Looks for cohorts of new accounts rating each other heavily within a short time.

8.7.6 Manipulation Rings

```
def find_manipulation_rings(self):
    Extract only strong positive edges (weight≥8)
   pos_G = nx.DiGraph([...])
    Undirected connected components size≥3
    For each, compute:
      – density, avg internal vs external rating, reciprocity
    Flag those with density>0.5, avg_rating>8, high reciprocity, large rating_diff.
    Visualize the densest ring.
```

Catches small cliques artificially inflating each other’s reputations.

8.8 Orchestration and Reporting

```
def run_full_analysis(self):
   self.load_data()
   self.build_gnn_model()
   results = {
      'anomalous_clusters': self.detect_anomalous_clusters(),
      'temporal_anomalies':  self.detect_temporal_anomalies(),
      'trust_asymmetry':     self.detect_trust_asymmetry(),
      'negative_subgraphs':  self.detect_negative_subgraphs(),
      'sybil_attacks':       self.detect_sybil_attacks(),
      'manipulation_rings':  self.find_manipulation_rings()
   }
```

```
    Print a concise summary of each detected anomaly.
    return results

def run_bitcoin_alpha_analysis(filepath):
    detector = AnomalousTrustDetector(filepath)
    return detector.run_full_analysis()
```

`run_full_analysis` ties all steps together, returning a dictionary of DataFrames or lists for each anomaly type.

The `__main__` block invokes `run_bitcoin_alpha_analysis` on the provided CSV path.

9. Conclusion

The Bitcoin Alpha trust network demonstrates complex social dynamics, including evidence of potential rating manipulation, retaliatory behavior, and the formation of insular trust clusters. While the majority of interactions (93.6%) generate positive ratings, the identified anomalies suggest vulnerabilities within the system that could undermine its reliability as a trust mechanism.

The temporal patterns reveal a platform that experienced significant growth and engagement followed by declining activity, with periodic anomalous events throughout its history. These findings highlight the challenges inherent in maintaining distributed trust systems and suggest that ongoing vigilance and advanced anomaly detection are critical for ensuring the integrity of decentralized reputation mechanisms.