Model 3: Graph-Based Campaign Mapping with Neo4j (Hours 15-16)

Team Member 4 Focus

```python
```

```
from neo4j import GraphDatabase
import networkx as nx
import pandas as pd
from datetime import datetime

class ShadowTraceGraphAnalyzer:
    def __init__(self, uri="bolt://localhost:7687", user="neo4j", password="password"):
        self.driver = GraphDatabase.driver(uri, auth=(user, password))
        self.nx_graph = nx.Graph()

    def close(self):
        self.driver.close()

    def create_user_network(self, interactions_df):
        """Build network from Instagram interactions"""
        with self.driver.session() as session:
            # Create users
            for user_id in pd.concat([interactions_df['user1'], interactions_df['user2']]).unique():
                session.run(
                    "MERGE (u:User {id: $user_id})",
                    user_id=user_id
                )

            # Create interactions
            for _, row in interactions_df.iterrows():
                session.run("""
                    MATCH (u1:User {id: $user1}), (u2:User {id: $user2})
                    MERGE (u1)-[r:INTERACTS {
                        type: $interaction_type,
                        weight: $weight,
                        timestamp: $timestamp
                    }]->(u2)
                """,
                user1=row['user1'],
                user2=row['user2'],
                interaction_type=row.get('interaction_type', 'unknown'),
                weight=row.get('weight', 1),
                timestamp=row.get('timestamp', datetime.now().isoformat())
                )

    def detect_coordinated_campaigns(self):
        """Find suspicious coordinated behavior patterns"""
        with self.driver.session() as session:
            # Find clusters of accounts that interact frequently
            result = session.run("""
                MATCH (u1:User)-[r:INTERACTS]->(u2:User)
```

```
            WHERE r.weight > 5
            WITH u1, u2, r.weight as weight
            MATCH (u1)-[:INTERACTS]->(common)<-[:INTERACTS]-(u2)
            WITH u1, u2, weight, count(common) as mutual_connections
            WHERE mutual_connections > 3
            RETURN u1.id as user1, u2.id as user2, weight, mutual_connections
            ORDER BY weight DESC, mutual_connections DESC
            LIMIT 50
        """)

        suspicious_pairs = []
        for record in result:
            suspicious_pairs.append({
                'user1': record['user1'],
                'user2': record['user2'],
                'interaction_weight': record['weight'],
                'mutual_connections': record['mutual_connections']
            })

        return suspicious_pairs

    def find_fake_account_clusters(self, fake_accounts):
        """Identify clusters of fake accounts"""
        with self.driver.session() as session:
            fake_account_list = "', '".join(fake_accounts)
            result = session.run(f"""
                MATCH (fake:User)-[r:INTERACTS]-(other:User)
                WHERE fake.id IN ['{fake_account_list}']
                WITH fake, other, sum(r.weight) as total_weight
                WHERE total_weight > 2
                RETURN fake.id as fake_account, other.id as connected_account, total_weight
                ORDER BY total_weight DESC
            """)

            clusters = {}
            for record in result:
                fake_acc = record['fake_account']
                if fake_acc not in clusters:
                    clusters[fake_acc] = []
                clusters[fake_acc].append({
                    'connected_account': record['connected_account'],
                    'strength': record['total_weight']
                })

            return clusters

    def generate_campaign_report(self, target_user):
```

```
        """Generate comprehensive campaign analysis report"""
        with self.driver.session() as session:
            # Find all accounts targeting the user
            result = session.run("""
                MATCH (attacker:User)-[r:INTERACTS]->(target:User {id: $target})
                WHERE r.type IN ['mention', 'reply', 'tag']
                WITH attacker, target, sum(r.weight) as attack_intensity
                WHERE attack_intensity > 3
                RETURN attacker.id as attacker_id, attack_intensity
                ORDER BY attack_intensity DESC
                LIMIT 20
            """, target=target_user)

            attackers = []
            for record in result:
                attackers.append({
                    'user_id': record['attacker_id'],
                    'attack_intensity': record['attack_intensity']
                })

            # Find coordinated timing patterns
            timing_result = session.run("""
                MATCH (u:User)-[r:INTERACTS]->(target:User {id: $target})
                WHERE r.timestamp > datetime() - duration('PT24H')
                RETURN date(r.timestamp) as attack_date, count(*) as attack_count
                ORDER BY attack_date DESC
            """, target=target_user)

            timeline = []
            for record in timing_result:
                timeline.append({
                    'date': record['attack_date'],
                    'attack_count': record['attack_count']
                })

            return {
                'target_user': target_user,
                'top_attackers': attackers,
                'attack_timeline': timeline,
                'total_attackers': len(attackers),
                'analysis# 24-Hour Hackathon Guide: VIP Threat Detection System

## 🎯 **Priority Focus for 24 Hours**
Given time constraints, focus on these core AI models:
1. **Threat Detection (NLP)** - Highest priority
2. **Fake Account Detection** - Medium priority
3. **Basic Dashboard** - For demo
```

```
4. **Graph Analysis** - If time permits

---

## **PHASE 1: Setup & Data Preparation (Hours 1-4)**

### Hour 1: Environment Setup
```bash
# Create virtual environment
python -m venv vip_monitor
source vip_monitor/bin/activate  # Linux/Mac
# vip_monitor\Scripts\activate  # Windows

# Install core libraries
pip install pandas numpy scikit-learn matplotlib seaborn
pip install transformers torch
pip install nltk textblob
pip install networkx plotly
pip install streamlit
pip install requests beautifulsoup4
```

Hours 2-3: Data Collection Strategy

Since Instagram API has restrictions, prepare multiple data sources:

Option 1: Simulated Dataset (Recommended for Hackathon)

```
python
```
```

```python
# Create synthetic dataset with realistic patterns
import pandas as pd
import numpy as np
from datetime import datetime, timedelta

def create_synthetic_data():
    # Generate fake Instagram posts/comments with threat labels
    threat_keywords = ["hate", "kill", "destroy", "fake", "scam"]
    normal_keywords = ["great", "love", "amazing", "thanks", "cool"]

    data = []
    for i in range(1000):
        is_threat = np.random.choice([0, 1], p=[0.8, 0.2])
        keywords = threat_keywords if is_threat else normal_keywords
        text = f"This is a {np.random.choice(keywords)} post about celebrity"

        data.append({
            'post_id': f'post_{i}',
            'text': text,
            'user_id': f'user_{np.random.randint(1, 200)}',
            'followers_count': np.random.randint(10, 10000),
            'following_count': np.random.randint(50, 5000),
            'post_count': np.random.randint(1, 1000),
            'account_age_days': np.random.randint(1, 1000),
            'is_threat': is_threat,
            'timestamp': datetime.now() - timedelta(days=np.random.randint(0, 30))
        })

    return pd.DataFrame(data)
```

Option 2: Web Scraping (Use Carefully)

python

```python
import requests
from bs4 import BeautifulSoup
import time

def scrape_public_data():
    # Only scrape publicly available, non-protected content
    # Add delays, respect robots.txt
    pass
```

Hour 4: Data Preprocessing

python

```python
def preprocess_text(text):
    import re
    import nltk
    nltk.download('stopwords')
    from nltk.corpus import stopwords

    # Clean text
    text = re.sub(r'http\S+', '', text)  # Remove URLs
    text = re.sub(r'@\w+', '', text)     # Remove mentions
    text = re.sub(r'#\w+', '', text)     # Remove hashtags
    text = re.sub(r'[^a-zA-Z\s]', '', text)  # Remove special chars

    # Convert to lowercase
    text = text.lower().strip()

    return text
```

PHASE 2: AI Model Development (Hours 5-16)

Model 1: Threat Detection (Hours 5-10)

Hour 5-6: Quick NLP Model

```python
```

```python
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix

class ThreatDetector:
    def __init__(self):
        self.vectorizer = TfidfVectorizer(max_features=5000, stop_words='english')
        self.model = LogisticRegression()

    def train(self, texts, labels):
        X = self.vectorizer.fit_transform(texts)
        X_train, X_test, y_train, y_test = train_test_split(X, labels, test_size=0.2)

        self.model.fit(X_train, y_train)

        # Evaluate
        y_pred = self.model.predict(X_test)
        print(classification_report(y_test, y_pred))

        return self.model

    def predict(self, text):
        X = self.vectorizer.transform([text])
        prob = self.model.predict_proba(X)[0][1]  # Probability of threat
        return prob > 0.5, prob
```

Hours 7-10: Advanced NLP with Transformers

python

```python
from transformers import AutoTokenizer, AutoModelForSequenceClassification
from transformers import Trainer, TrainingArguments
import torch

class AdvancedThreatDetector:
    def __init__(self):
        self.model_name = "distilbert-base-uncased"
        self.tokenizer = AutoTokenizer.from_pretrained(self.model_name)
        self.model = AutoModelForSequenceClassification.from_pretrained(
            self.model_name, num_labels=2
        )

    def prepare_data(self, texts, labels):
        encodings = self.tokenizer(
            texts,
            truncation=True,
            padding=True,
            max_length=128,
            return_tensors='pt'
        )

        class Dataset(torch.utils.data.Dataset):
            def __init__(self, encodings, labels):
                self.encodings = encodings
                self.labels = labels

            def __getitem__(self, idx):
                item = {key: torch.tensor(val[idx]) for key, val in self.encodings.items()}
                item['labels'] = torch.tensor(self.labels[idx])
                return item

            def __len__(self):
                return len(self.labels)

        return Dataset(encodings, labels)

    def train_model(self, train_texts, train_labels, val_texts, val_labels):
        train_dataset = self.prepare_data(train_texts, train_labels)
        val_dataset = self.prepare_data(val_texts, val_labels)

        training_args = TrainingArguments(
            output_dir='./threat_model',
            num_train_epochs=3,
            per_device_train_batch_size=16,
            per_device_eval_batch_size=64,
            warmup_steps=500,
```

```
        weight_decay=0.01,
        logging_dir='./logs',
    )

    trainer = Trainer(
        model=self.model,
        args=training_args,
        train_dataset=train_dataset,
        eval_dataset=val_dataset,
    )

    trainer.train()
    return trainer
```

Model 2: Fake Account Detection (Hours 11-14)

python

```python
from sklearn.ensemble import IsolationForest
from sklearn.preprocessing import StandardScaler

class FakeAccountDetector:
    def __init__(self):
        self.scaler = StandardScaler()
        self.model = IsolationForest(contamination=0.1, random_state=42)

    def extract_features(self, df):
        """Extract features indicating fake accounts"""
        features = pd.DataFrame()

        # Follower-to-following ratio
        features['follower_following_ratio'] = df['followers_count'] / (df['following_count'] + 1)

        # Posts per day since account creation
        features['posts_per_day'] = df['post_count'] / (df['account_age_days'] + 1)

        # Account age
        features['account_age'] = df['account_age_days']

        # Profile completeness (simulate)
        features['has_profile_pic'] = np.random.choice([0, 1], size=len(df), p=[0.1, 0.9])
        features['has_bio'] = np.random.choice([0, 1], size=len(df), p=[0.2, 0.8])

        # Suspicious patterns
        features['very_new_account'] = (df['account_age_days'] < 30).astype(int)
        features['high_following_ratio'] = (df['following_count'] > df['followers_count'] * 3).astype(int)

        return features

    def train(self, df):
        features = self.extract_features(df)
        X_scaled = self.scaler.fit_transform(features)
        self.model.fit(X_scaled)
        return self.model

    def predict(self, df):
        features = self.extract_features(df)
        X_scaled = self.scaler.transform(features)
        predictions = self.model.predict(X_scaled)
        scores = self.model.decision_function(X_scaled)

        # Convert to probability-like scores
```

```
    fake_prob = (1 - (scores + 1) / 2)  # Normalize to 0-1
    return predictions == -1, fake_prob  # -1 means outlier (fake)
```

Model 3: Basic Graph Analysis (Hours 15-16)

python

```python
import networkx as nx
import matplotlib.pyplot as plt

class CampaignMapper:
    def __init__(self):
        self.graph = nx.Graph()

    def build_network(self, interactions_df):
        """Build network from user interactions"""
        # interactions_df should have columns: user1, user2, interaction_type, weight
        for _, row in interactions_df.iterrows():
            self.graph.add_edge(
                row['user1'],
                row['user2'],
                weight=row.get('weight', 1),
                type=row.get('interaction_type', 'unknown')
            )

    def detect_communities(self):
        """Detect coordinated groups"""
        from networkx.algorithms import community
        communities = community.greedy_modularity_communities(self.graph)
        return list(communities)

    def find_suspicious_clusters(self, min_size=5):
        """Find clusters that might be coordinated campaigns"""
        communities = self.detect_communities()
        suspicious = []

        for community in communities:
            if len(community) >= min_size:
                # Calculate cluster metrics
                subgraph = self.graph.subgraph(community)
                density = nx.density(subgraph)
                avg_clustering = nx.average_clustering(subgraph)

                if density > 0.3 and avg_clustering > 0.5:  # Thresholds for suspicious behavior
                    suspicious.append({
                        'nodes': list(community),
                        'size': len(community),
                        'density': density,
                        'clustering': avg_clustering
                    })

        return suspicious
```

PHASE 3: Integration & Dashboard (Hours 17-22)

Hours 17-20: Streamlit Dashboard

python

```python
import streamlit as st
import plotly.express as px
import plotly.graph_objects as go

def create_dashboard():
    st.title("🛡️ VIP Threat Monitoring System")

    # Sidebar for controls
    st.sidebar.header("Monitoring Controls")
    target_vip = st.sidebar.text_input("Target VIP Username", "celebrity_name")
    threat_threshold = st.sidebar.slider("Threat Threshold", 0.0, 1.0, 0.7)

    # Main dashboard
    col1, col2, col3, col4 = st.columns(4)

    with col1:
        st.metric("Active Threats", "23", "↑ 5")
    with col2:
        st.metric("Fake Accounts", "156", "↓ 12")
    with col3:
        st.metric("Coordinated Campaigns", "3", "→ 0")
    with col4:
        st.metric("Risk Score", "HIGH", "↑")

    # Real-time threat feed
    st.header("🚨 Real-Time Threat Detection")

    # Simulate real-time data
    threat_data = generate_sample_threats()

    for threat in threat_data[:10]:  # Show top 10
        with st.expander(f"Threat Score: {threat['score']:.2f} - {threat['type']}"):
            st.write(f"**User:** {threat['user']}")
            st.write(f"**Content:** {threat['content']}")
            st.write(f"**Timestamp:** {threat['timestamp']}")
            st.write(f"**Evidence:** {threat['evidence']}")

            col1, col2, col3 = st.columns(3)
            with col1:
                st.button("✅ Confirm Threat", key=f"confirm_{threat['id']}")
            with col2:
                st.button("❌ False Positive", key=f"false_{threat['id']}")
            with col3:
                st.button("📊 Investigate", key=f"invest_{threat['id']}")

    # Visualization section
```

```
    st.header("📊 Threat Analytics")

    # Threat timeline
    fig_timeline = px.line(
        x=[1, 2, 3, 4, 5],
        y=[10, 15, 13, 17, 22],
        title="Threats Over Time"
    )
    st.plotly_chart(fig_timeline)

    # Network visualization
    st.header("🕸️ Campaign Network")
    # Add network visualization here

def generate_sample_threats():
    """Generate sample threat data for demo"""
    threats = []
    for i in range(20):
        threats.append({
            'id': i,
            'score': np.random.uniform(0.7, 1.0),
            'type': np.random.choice(['Death Threat', 'Harassment', 'Impersonation', 'Misinformation']),
            'user': f'suspicious_user_{i}',
            'content': f'This is a sample threatening message {i}',
            'timestamp': datetime.now() - timedelta(minutes=np.random.randint(1, 60)),
            'evidence': f'Pattern match, sentiment analysis, user behavior'
        })
    return sorted(threats, key=lambda x: x['score'], reverse=True)

if __name__ == "__main__":
    create_dashboard()
```

Hours 21-22: API Integration

python

```python
from fastapi import FastAPI, BackgroundTasks
from pydantic import BaseModel
import uvicorn

app = FastAPI(title="VIP Threat Monitoring API")

class ThreatAlert(BaseModel):
    user_id: str
    content: str
    threat_score: float
    threat_type: str
    evidence: list

# Initialize models
threat_detector = ThreatDetector()
fake_detector = FakeAccountDetector()

@app.post("/analyze/threat")
async def analyze_threat(content: str):
    is_threat, score = threat_detector.predict(content)
    return {
        "is_threat": is_threat,
        "threat_score": float(score),
        "status": "processed"
    }

@app.post("/analyze/account")
async def analyze_account(account_data: dict):
    # Process account data
    df = pd.DataFrame([account_data])
    is_fake, fake_score = fake_detector.predict(df)

    return {
        "is_fake": bool(is_fake[0]),
        "fake_score": float(fake_score[0]),
        "status": "processed"
    }

@app.get("/dashboard/stats")
async def get_dashboard_stats():
    return {
        "active_threats": 23,
        "fake_accounts": 156,
        "campaigns": 3,
```

```
    "risk_level": "HIGH"
}
```

PHASE 4: Final Demo Preparation (Hours 23-24)

Hour 23: Testing & Bug Fixes

- Test all models with sample data
- Fix any critical bugs
- Prepare demo scenarios

Hour 24: Presentation Prep

```python
```

```python
# Create a comprehensive demo script
def run_demo():
    print("🛡️ VIP Threat Monitoring System Demo")
    print("=" * 50)

    # Demo 1: Threat Detection
    sample_threats = [
        "I hate this celebrity, they should disappear forever",
        "Great performance last night! Loved the show",
        "This fake account is spreading lies about the star"
    ]

    print("\n1. THREAT DETECTION:")
    for text in sample_threats:
        is_threat, score = threat_detector.predict(text)
        print(f"Text: {text[:50]}...")
        print(f"Threat: {'YES' if is_threat else 'NO'} (Score: {score:.3f})")
        print("-" * 30)

    # Demo 2: Fake Account Detection
    print("\n2. FAKE ACCOUNT DETECTION:")
    sample_accounts = pd.DataFrame([
        {'followers_count': 50000, 'following_count': 50000, 'post_count': 5, 'account_age_days': 7},
        {'followers_count': 1000, 'following_count': 1500, 'post_count': 200, 'account_age_days': 365}
    ])

    is_fake, fake_scores = fake_detector.predict(sample_accounts)
    for i, (fake, score) in enumerate(zip(is_fake, fake_scores)):
        print(f"Account {i+1}: {'FAKE' if fake else 'REAL'} (Score: {score:.3f})")

    print("\n3. Starting Dashboard...")
    print("Run: streamlit run dashboard.py")
```

🎯 Hackathon Success Tips

Priority Order:

1. ✅ Get threat detection working (80% accurate is fine)
2. ✅ Create basic dashboard with sample data
3. ✅ Add fake account detection
4. ✅ Polish the demo presentation
5. ⭐ Bonus: Network analysis if time permits

Demo Strategy:

- **Start with the problem** - Show real examples of threats
- **Show the solution** - Live demo of threat detection
- **Highlight AI sophistication** - Mention transformers, anomaly detection
- **Show business impact** - Time saved, threats prevented
- **End with scalability** - How it can protect multiple VIPs

Technical Backup Plans:

- If advanced models fail, fall back to simple ML
- If real data is unavailable, use convincing synthetic data
- If dashboard breaks, use Jupyter notebooks for demo
- Always have screenshots/videos as backup

Presentation Points:

- **Innovation**: Real-time AI-driven threat detection
- **Technical Depth**: Multiple ML models, graph analysis
- **Practical Impact**: Protects VIPs, saves investigation time
- **Scalability**: Can monitor multiple accounts simultaneously
- **Integration Ready**: APIs for law enforcement tools

🚀 Quick Start Commands

```bash
# Setup
python -m venv vip_env
source vip_env/bin/activate
pip install -r requirements.txt

# Train models
python train_models.py

# Start dashboard
streamlit run dashboard.py

# Start API
uvicorn main:app --reload
```

Remember: Focus on getting a working demo rather than perfect accuracy. Judges value innovation, technical depth, and practical impact over perfect metrics!