AICS Lesson 9: AI for User Behavior Analytics (UBA) - Code Demonstration Guide

Technical Implementation and Hands-On Practice

Course: AI in Cybersecurity - Class 09
Focus: Understanding UBA through practical implementation

🎯 Learning Objectives

By working through this code demonstration, you will:

- ✅ **Understand** how to generate and preprocess user behavior data
- ✅ **Implement** baseline behavior modeling using statistical methods
- ✅ **Apply** multiple anomaly detection algorithms (statistical and ML-based)
- ✅ **Build** insider threat and account takeover detection systems
- ✅ **Create** comprehensive risk scoring mechanisms
- ✅ **Interpret** UBA dashboard visualizations and results

🔧 Technical Prerequisites

Required Python Libraries:

```
import pandas as pd              # Data manipulation and analysis
import numpy as np               # Numerical computations
import matplotlib.pyplot as plt  # Basic plotting
import seaborn as sns           # Advanced statistical visualizations
from datetime import datetime, timedelta  # Date/time handling

# Machine Learning Libraries
from sklearn.ensemble import IsolationForest     # Anomaly detection
from sklearn.cluster import DBSCAN              # Density-based clustering
from sklearn.preprocessing import StandardScaler # Feature scaling
from sklearn.decomposition import PCA           # Dimensionality reduction
import scipy.stats as stats                     # Statistical functions
```

Environment Setup:

- **Python Version:** 3.8 or higher
- **Jupyter Notebook** or **Google Colab** recommended
- **Memory:** At least 4GB RAM for larger datasets
- **Storage:** 100MB for code and generated data

📊 Section 1: Data Generation and Understanding

1.1 Synthetic User Behavior Data

What the Code Does: The `generate_user_behavior_data()` function creates realistic user behavior patterns that simulate:

User Types and Their Behaviors:

- **Developer:** Heavy application usage, irregular hours, high data access
- **Analyst:** Regular business hours, moderate data access, reporting tools
- **Manager:** Light application usage, standard hours, executive system access
- **Admin:** System administration tools, off-hours maintenance, high privileges
- **Contractor:** Limited access patterns, restricted system usage

Key Features Generated:

```
# Behavioral Features in the Dataset
'login_hour'            # Time of day user logs in (0-23)
'logout_hour'           # Time of day user logs out (0-23)
'session_duration'      # How long user stays logged in (hours)
'files_accessed'        # Number of files accessed per session
'failed_logins'         # Number of failed login attempts
'sensitive_files_accessed' # Access to classified/sensitive data
'applications_used'     # Number of different applications used
'unique_ips'            # Number of different IP addresses used
'location'              # Geographic location of access
'is_weekend'            # Boolean - whether activity occurred on weekend
```

Normal vs. Anomalous Behavior Simulation:

- **95% Normal Behavior:** Follows user's established patterns
- **5% Anomalous Behavior:** Introduces suspicious activities:
    - Unusual login times (2 AM, 11 PM)
    - Excessive data access (100+ files instead of usual 20)

- Suspicious locations (Moscow, Beijing, Unknown)
- High failed login rates (5+ instead of usual 0.5)

1.2 Understanding the Data

Practice Exercise 1:

```
# Explore the generated dataset
print("Dataset Shape:", df.shape)
print("\nColumn Information:")
print(df.info())
print("\nSample Records:")
print(df.head())
print("\nStatistical Summary:")
print(df.describe())
```

Questions to Consider:

1. How many users and total records were generated?
2. What's the range of login hours across all users?
3. Which user type shows the highest variance in file access?
4. How does weekend activity compare to weekday activity?

📈 Section 2: Baseline Behavior Modeling

2.1 Individual User Baselines

Concept: Each user has unique behavioral patterns that AI learns automatically.

Key Statistical Measures:

- **Mean (Average):** Central tendency of user's behavior
- **Standard Deviation:** Variability in user's patterns
- **Mode:** Most frequent behavior (for categorical data like locations)

Implementation Details:

```
def create_user_baselines(df):

    # For each user, calculate:
```

```
    baseline = {
        'avg_login_hour': user_data['login_hour'].mean(),
        'std_login_hour': user_data['login_hour'].std(),
        'avg_session_duration': user_data['session_duration'].mean(),
        'std_session_duration': user_data['session_duration'].std(),
        # ... more statistical measures
    }
```

Practice Exercise 2:

```
# Analyze a specific user's baseline
user_id = 'user_001'
user_data = df[df['user_id'] == user_id]
print(f"User {user_id} Baseline Analysis:")
print(f"Average login hour: {user_data['login_hour'].mean():.2f}")
print(f"Login hour std dev: {user_data['login_hour'].std():.2f}")
print(f"Typical session duration:
{user_data['session_duration'].mean():.2f} hours")
print(f"Common locations: {user_data['location'].mode().tolist()}")
```

2.2 Peer Group Analysis

Concept: Compare individual behavior to others in similar roles to identify outliers.

Why It Matters:

- Identifies users behaving differently from their peers
- Accounts for role-based behavioral differences
- Helps distinguish legitimate role changes from suspicious activity

Implementation:

```
# Compare user to peer group
user_type = 'developer'
peer_data = df[df['user_type'] == user_type]
individual_data = df[df['user_id'] == 'user_001']
print("Peer Group vs. Individual Analysis:")
print(f"Peer avg files accessed: {peer_data['files_accessed'].mean():.2f}")
print(f"Individual avg files:
{individual_data['files_accessed'].mean():.2f}")
```

🚨 Section 3: Anomaly Detection Techniques

3.1 Statistical Anomaly Detection (Z-Score Method)

Concept: Identify activities that deviate significantly from established baselines.

Z-Score Formula:

$Z = (X - \mu) / \sigma$

Where:

X = observed value

μ = mean of baseline

σ = standard deviation of baseline

Interpretation:

- **$Z < 2$:** Normal behavior
- **$2 \leq Z < 3$:** Moderately unusual (investigate)
- **$Z \geq 3$:** Highly anomalous (alert)

Code Implementation:

```
def detect_statistical_anomalies(df, user_baselines, z_threshold=2.5):

    # For each user activity:
    z_login = abs((row['login_hour'] - baseline['avg_login_hour']) /
baseline['std_login_hour'])
    z_files = abs((row['files_accessed'] - baseline['avg_files_accessed'])
/ baseline['std_files_accessed'])

    # Flag as anomaly if any Z-score exceeds threshold
    is_anomaly = max(z_login, z_files) > z_threshold
```

Practice Exercise 3:

```
# Calculate Z-scores for a specific activity
user_baseline = user_baselines['user_001']
suspicious_activity = {
```

```
    'login_hour': 2.0,  # 2 AM login
    'files_accessed': 150  # Much higher than normal
}
z_hour = abs((suspicious_activity['login_hour'] -
user_baseline['avg_login_hour']) /
             user_baseline['std_login_hour'])
z_files = abs((suspicious_activity['files_accessed'] -
user_baseline['avg_files_accessed']) /
              user_baseline['std_files_accessed'])
print(f"Z-score for login hour: {z_hour:.2f}")
print(f"Z-score for files accessed: {z_files:.2f}")
print(f"Is anomalous: {max(z_hour, z_files) > 2.5}")
```

3.2 Machine Learning Anomaly Detection (Isolation Forest)

Concept: ML algorithm that isolates anomalies by randomly selecting features and split values.

How Isolation Forest Works:

1. **Random Sampling:** Select random subset of data
2. **Random Splits:** Create binary trees with random feature splits
3. **Isolation Measure:** Anomalies require fewer splits to isolate
4. **Scoring:** Calculate anomaly score based on path length

Advantages:

- No need for labeled training data (unsupervised)
- Effective with high-dimensional data
- Scales well to large datasets
- Handles multiple features simultaneously

Code Implementation:

```
# Prepare features for ML analysis
feature_columns = ['login_hour', 'session_duration', 'files_accessed',
                   'failed_logins', 'sensitive_files_accessed']

# Apply Isolation Forest
iso_forest = IsolationForest(contamination=0.1, random_state=42)
anomaly_labels = iso_forest.fit_predict(X_scaled)  # -1 = anomaly, 1 =
normal
anomaly_scores = iso_forest.decision_function(X_scaled)  # Continuous
scores
```

Practice Exercise 4:

```
# Analyze ML anomaly detection results
ml_anomalies = df_with_ml[df_with_ml['is_anomaly_ml']]
print(f"Total anomalies detected: {len(ml_anomalies)}")
print(f"Percentage of data flagged: {len(ml_anomalies)/len(df)*100:.2f}%")

# Look at most anomalous activities
most_anomalous = df_with_ml.nsmallest(5, 'anomaly_score_ml')
print("\nMost anomalous activities:")
print(most_anomalous[['user_id', 'login_hour', 'files_accessed',
'location', 'anomaly_score_ml']])
```

🔍 Section 4: Insider Threat Detection

4.1 Threat Indicators

Key Behavioral Indicators the Code Detects:

1. **Excessive Sensitive File Access**

```
recent_sensitive = user_data.tail(7)['sensitive_files_accessed'].mean()
if recent_sensitive > baseline['avg_sensitive_access'] + 3 *
baseline['std_sensitive_access']:

    # Flag as potential data exfiltration
```

2. **Unusual Working Hours Pattern**

```
after_hours = user_data[(user_data['login_hour'] < 6) |
(user_data['login_hour'] > 20)]
if len(after_hours) > len(user_data) * 0.3:  # More than 30% after hours

    # Flag as suspicious schedule change
```

3. **Data Exfiltration Pattern**

```
# High file access combined with failed logins suggests credential attacks
high_access_days = user_data[user_data['files_accessed'] > threshold]
if high_access_days['failed_logins'].mean() > baseline['avg_failed_logins']
* 2:

    # Flag as potential data theft attempt
```

4.2 Risk Scoring Algorithm

Threat Score Calculation:

```
# Calculate overall insider threat score
threat_score = sum(risk_indicators.values()) / len(risk_indicators)

# Risk Levels:
# 0-1: Low Risk
# 1-2: Medium Risk
# 2-3: High Risk
# 3+: Critical Risk
```

Practice Exercise 5:

```
# Analyze insider threat detection results
for threat in insider_threats[:3]:
    print(f"User: {threat['user_id']}")
    print(f"Threat Score: {threat['threat_score']:.2f}")
    print(f"Risk Indicators: {threat['risk_indicators']}")
    print("---")
```

🔐 Section 5: Account Takeover Detection

5.1 ATO Detection Algorithms

Key Detection Methods:

1. **Geographic Anomalies**

```
# Detect logins from unusual locations
recent_locations = recent_data['location'].unique()
unusual_locations = [loc for loc in recent_locations if loc not in
baseline_locations]
```

2. **Temporal Anomalies**

```
# Detect significant changes in login time patterns
recent_avg_login = recent_data['login_hour'].mean()
baseline_avg_login = baseline['avg_login_hour']
time_shift = abs(recent_avg_login - baseline_avg_login)
```

3. **Failed Login Spikes**

```
# Multiple failed logins followed by successful access
high_failed_login_days = recent_data[recent_data['failed_logins'] >
threshold]
```

5.2 Risk Assessment

ATO Risk Score Components:

- **New Locations:** +2 points per unusual location
- **Login Time Shift:** +0.5 points per hour difference
- **Failed Login Spikes:** +0.2 points per failed attempt above baseline
- **Multiple IPs:** +0.2 points per unique IP above threshold

Practice Exercise 6:

```
# Analyze ATO detection results
for alert in ato_alerts[:3]:
    print(f"User: {alert['user_id']}")
    print(f"ATO Risk Score: {alert['ato_risk_score']:.2f}")
    print(f"Indicators: {list(alert['indicators'].keys())}")
    print(f"Last Activity: {alert['last_activity']}")
    print("---")
```

📊 Section 6: Comprehensive Risk Scoring

6.1 Multi-Factor Risk Assessment

Combined Risk Score Formula:

```
total_score = (
    statistical_anomaly_score * 0.3 +     # 30% weight
    insider_threat_score * 0.4 +          # 40% weight
    ato_risk_score * 0.3                  # 30% weight
)
```

Risk Level Classification:

- **0-2:** LOW (Green) - Normal activity
- **2-5:** MEDIUM (Yellow) - Monitor closely
- **5-8:** HIGH (Orange) - Investigate immediately
- **8+:** CRITICAL (Red) - Immediate response required

6.2 Risk Score Interpretation

Practice Exercise 7:

```
# Analyze comprehensive risk scores
high_risk_users = [(user, scores) for user, scores in risk_scores.items()
                   if scores['total_risk_score'] >= 5]
print(f"High-risk users identified: {len(high_risk_users)}")
for user_id, scores in high_risk_users:
    user_type = df[df['user_id'] == user_id]['user_type'].iloc[0]
    print(f"{user_id} ({user_type}): {scores['total_risk_score']:.2f} -
{scores['risk_level']}")
```

📈 Section 7: Dashboard Visualization Interpretation

7.1 Understanding the Charts

Chart 1: Risk Score Distribution

- **Purpose:** Shows how risk scores are spread across the user population
- **Good Pattern:** Most users should have low scores (left-skewed distribution)
- **Warning Signs:** Too many high scores or flat distribution

Chart 2: Risk Level Pie Chart

- **Healthy Organization:** >60% Low risk, <5% Critical risk
- **Investigation Needed:** >20% High/Critical risk users

Chart 3: Detection Method Comparison

- **Analysis:** Compare effectiveness of different detection algorithms
- **Tuning Guidance:** High ML anomalies with low insider threats may indicate over-sensitivity

7.2 Dashboard Analysis Skills

Practice Exercise 8:

```
# Create your own analysis
risk_distribution = {}
for scores in risk_scores.values():
    level = scores['risk_level']
    risk_distribution[level] = risk_distribution.get(level, 0) + 1
print("Risk Distribution Analysis:")
for level in ['LOW', 'MEDIUM', 'HIGH', 'CRITICAL']:
    count = risk_distribution.get(level, 0)
    percentage = (count / len(risk_scores)) * 100
    print(f"{level}: {count} users ({percentage:.1f}%)")
```

🛠️ Hands-On Exercises

Exercise 1: Modify User Behavior Patterns

Task: Create a new user type with specific behavioral characteristics.

```
# Add your code here to create a "Security Analyst" user type

# Consider: What would their normal patterns look like?

# - Login times?

# - Applications used?

# - File access patterns?
```

Exercise 2: Adjust Detection Sensitivity

Task: Experiment with different threshold values and observe changes in detection rates.

```
# Try different Z-score thresholds
for threshold in [2.0, 2.5, 3.0]:
    anomalies = detect_statistical_anomalies(df, user_baselines, threshold)
    print(f"Threshold {threshold}: {len(anomalies)} anomalies detected")
```

Exercise 3: Custom Risk Indicator

Task: Add a new risk indicator for insider threat detection.

```
# Example: Detect users accessing systems outside their department
def detect_cross_department_access(user_data, user_type):
    # Your implementation here
    pass
```

Exercise 4: Build a Simple Alert System

Task: Create a function that generates prioritized alerts based on risk scores.

```
def generate_security_alerts(risk_scores, threshold=5.0):
    alerts = []
    # Your implementation here
    return sorted(alerts, key=lambda x: x['priority'], reverse=True)
```

🔬 Advanced Analysis Questions

Technical Understanding:

1. **Why does the code use both statistical and ML-based anomaly detection?**
    - Hint: Consider the strengths and weaknesses of each approach
2. **How does the Isolation Forest algorithm identify anomalies?**
    - Research: What makes it different from clustering-based methods?
3. **What role does feature scaling play in the ML anomaly detection?**
    - Experiment: Try running without StandardScaler and observe results

Practical Application:

4. **How would you handle seasonal patterns in user behavior?**
    - Consider: Year-end activities, holiday schedules, business cycles
5. **What additional features would improve detection accuracy?**

    - Think: Email patterns, document types, network traffic

6. **How would you reduce false positives in a real deployment?**

    - Consider: Whitelisting, contextual factors, analyst feedback

System Design:

7. **How would you scale this system for 10,000+ users?**

    - Think: Data storage, processing speed, memory requirements

8. **What privacy protections would you implement?**

    - Consider: Data anonymization, access controls, audit trails

📚 Code Enhancement Projects

Project 1: Real-Time Processing Simulation

Modify the code to simulate real-time behavioral analysis:

- Process data in streaming fashion
- Update baselines continuously
- Generate alerts in real-time

Project 2: Advanced Visualization Dashboard

Create interactive visualizations:

- User behavior timelines
- Geographic access maps
- Risk score trends over time

Project 3: Multi-Modal Behavioral Analysis

Expand the feature set:

- Email communication patterns
- File type access preferences
- Application usage sequences

Project 4: Automated Response System

Build automated response capabilities:

- Account suspension for high-risk activities
- MFA challenges for suspicious logins
- Escalation workflows for critical alerts

✅ Knowledge Check

After working through this code demonstration, you should be able to:

- [ ] **Generate** realistic user behavior datasets for UBA analysis
- [ ] **Calculate** statistical baselines for individual users and peer groups
- [ ] **Implement** both statistical and ML-based anomaly detection
- [ ] **Build** insider threat detection algorithms with multiple indicators
- [ ] **Create** account takeover detection systems
- [ ] **Design** comprehensive risk scoring mechanisms
- [ ] **Interpret** UBA dashboard visualizations and results
- [ ] **Modify** detection algorithms to reduce false positives
- [ ] **Explain** how different AI techniques complement each other in UBA

Next Steps:

1. **Experiment** with different parameter values and thresholds
2. **Research** real-world UBA implementations and compare approaches
3. **Practice** with your own behavioral data (email patterns, web usage)
4. **Explore** advanced ML techniques for behavioral analysis

🔗 Integration with Lesson Concepts

This code demonstration directly implements the concepts from the lesson:

- **Section 1-2** → Data sources and normalization from UBA architecture
- **Section 3** → Normal vs. anomalous behavior detection
- **Section 4** → Insider threat detection techniques
- **Section 5** → Account takeover detection methods
- **Section 6** → Risk scoring and alerting systems
- **Section 7** → Investigation and response dashboards

Understanding both the theoretical concepts AND the practical implementation gives you comprehensive UBA knowledge for real-world cybersecurity applications.

Remember: UBA is both an art and a science. The algorithms provide the foundation, but successful implementation requires understanding business context, user privacy, and organizational culture.