

# MACHINE LEARNING CYBER SECURITY

THREAT ROUTE 66



# **Using Machine Learning at Cyber Security**

**Threat Route 66**

**2025**



# Using Machine Learning at Cyber Security

## Table of Contents

---



### Course Overview

This comprehensive training course teaches machine learning techniques specifically for security and threat hunting applications. Through hands-on exercises and real-world examples, you'll learn to build, deploy, and maintain ML systems for cybersecurity.

### Course Structure

- **Duration:** 40+ hours of content
  - **Format:** Theory + Hands-on labs
  - **Level:** Beginner to Advanced
  - **Focus:** Security and threat detection applications
- 



### Prerequisites and Setup

#### Required Knowledge

- **Python Basics:** Variables, functions, loops, basic data structures
- **Command Line:** Basic terminal/command prompt usage
- **Mathematics:** High school level (algebra, basic statistics)
- **Security Fundamentals:** Basic understanding of network security concepts

#### System Requirements

##### Hardware

- **Minimum:**
  - 8GB RAM
  - 50GB free disk space
  - Dual-core processor
- **Recommended:**
  - 16GB+ RAM
  - 100GB+ free disk space (for datasets)

- Quad-core processor or better
- NVIDIA GPU (optional, for deep learning acceleration)

## Software Setup

```
# 1. Install Python (3.8 or higher)
# Download from python.org or use package manager

# 2. Create virtual environment
python -m venv ml_security_env

# 3. Activate environment
# Windows:
ml_security_env\Scripts\activate
# Linux/Mac:
source ml_security_env/bin/activate

# 4. Install required packages
pip install -r requirements.txt
```

## Requirements File (requirements.txt)

```
# Core Libraries
numpy==1.23.5
pandas==1.5.3
matplotlib==3.6.3
seaborn==0.12.2
scipy==1.10.0

# Machine Learning
scikit-learn==1.2.0
tensorflow==2.11.0
keras==2.11.0

# Deep Learning Support
opencv-python==4.7.0
pillow==9.4.0

# Data Processing
beautifulsoup4==4.11.1
requests==2.28.2
pymongo==4.3.3
sqlalchemy==2.0.0
```

```
# Visualization
plotly==5.13.0
wordcloud==1.9.2

# Utilities
tqdm==4.64.1
joblib==1.2.0
python-dotenv==0.21.1

# Optional (for genetic algorithms)
deap==1.3.3

# Jupyter Support
jupyter==1.0.0
ipykernel==6.20.2
```

Create a `data` directory and download the following datasets:

1. **Network Traffic Data**
    - [KDD Cup 99](#)
    - [CICIDS2017](#)
  2. **Malware Samples** (Handle with care!)
    - [MalwareBazaar](#) (API access)
    - [VirusShare](#) (Registration required)
  3. **Log Data**
    - [Logs from BlueTeam Labs](#)
    - Synthetic data generators included in course
- 

## Detailed Course Contents

### Section 1: Data Acquisition, Cleaning, and Manipulation

Duration: 6 hours | Difficulty: ★★☆☆☆

#### Topics Covered:

- Python data structures review
- NumPy arrays and operations
- Pandas DataFrames
- SQL database operations
- MongoDB and NoSQL basics
- Web scraping techniques
- Data cleaning strategies

#### Key Skills:

-  Extract data from multiple sources
-  Clean and preprocess security logs
-  Handle missing and corrupted data
-  Automate data collection pipelines

#### Labs:

1. Building a security log parser
2. Web scraping threat intelligence feeds
3. Creating a unified data pipeline

## Section 2: Data Exploration and Statistics

Duration: 6 hours | Difficulty: ★★★☆☆

### Topics Covered:

- Descriptive statistics
- Statistical distributions
- Hypothesis testing
- Probability theory
- Bayes' theorem
- Time series analysis
- Fourier transforms
- Anomaly detection with statistics

### Key Skills:

- Calculate security metrics
- Identify statistical anomalies
- Apply Bayesian inference
- Detect periodic patterns in logs

### Labs:

1. Statistical analysis of attack patterns
  2. Building a Bayesian spam filter
  3. Frequency analysis for DGA detection
  4. Time series anomaly detection
- 

## Section 3: Machine Learning Essentials - Trees, Forests, & K-Means

Duration: 8 hours | Difficulty: ★★★★★☆

### Topics Covered:

- Clustering algorithms (K-Means, DBSCAN, Hierarchical)
- K-Nearest Neighbors (KNN)

- Principal Component Analysis (PCA)
- Support Vector Machines (SVM)
- Decision Trees
- Random Forests
- Ensemble methods
- Model evaluation techniques

## Key Skills:

- Choose appropriate algorithms
- Implement clustering for threat grouping
- Build classification models
- Reduce dimensionality of security data
- Create ensemble anomaly detectors

## Labs:

1. Network traffic clustering
  2. Malware family classification
  3. User behavior profiling
  4. Multi-algorithm anomaly detection
- 

## Section 4: Deep Learning Essentials

Duration: 8 hours | Difficulty:

## Topics Covered:

- From linear to non-linear models
- Perceptron and neural network mathematics
- Activation functions (Sigmoid, ReLU, Softmax)
- Loss functions and their properties
- Gradient descent and backpropagation
- Regularization techniques (L1/L2, Dropout, BatchNorm)
- Multi-class classification with neural networks
- Real-time deployment considerations

## Key Concepts & Formulas:

- **Perceptron:**  $z = w^T x + b$ ,  $y = \sigma(z)$
- **Backpropagation:** Chain rule application

- **Memory Tricks:** WIBA (Weights, Inputs, Bias, Activation)
- **Memory Tricks:** FLEB (Forward, Loss, Error backward, Bias/weight update)

## Key Skills:

- Understand neural network mathematics
- Implement gradient descent from scratch
- Choose appropriate architectures and activations
- Apply regularization to prevent overfitting
- Deploy optimized real-time classifiers

## Labs:

1. Building a perceptron from scratch
  2. Multi-layer threat classifier with regularization
  3. Protocol identification system with softmax
  4. Real-time attack detection with optimization
- 

## Section 5: Autoencoders and Convolutional Networks

Duration: 8 hours | Difficulty: 

## Topics Covered:

- Understanding convolutions (1D and 2D)
- CNN building blocks (Conv, Pool, Dense)
- Receptive fields and feature hierarchies
- Embedding layers for discrete data
- Autoencoder fundamentals and mathematics
- Types of autoencoders (Vanilla, Sparse, Denoising, VAE)
- Reconstruction loss metrics
- Ensemble autoencoders for robust anomaly detection

## Key Concepts & Formulas:

- **Convolution:**  $(f * g)[n] = \sum f[m]g[n-m]$
- **Output Size:**  $\lfloor (in + 2 \times pad - kernel) / stride \rfloor + 1$
- **Reconstruction Loss:**  $MSE = (1/n) \sum \|x - \hat{x}\|^2$
- **Memory Tricks:** SLIP (Slide, Local, Invariant, Parameters)
- **Memory Tricks:** TED (Tokens to Embedding Dimensions)
- **Memory Tricks:** ELDER (Encode, Latent, Decode, Error, Reconstruct)

## Key Skills:

- Design CNNs for malware and text classification
- Build autoencoders for anomaly detection
- Implement embedding layers effectively
- Create ensemble models for robustness
- Deploy production anomaly detection systems

## Labs:

1. Malware image classification with CNN
  2. Spam detection using text CNNs
  3. Log anomaly detection with autoencoders
  4. Network traffic autoencoder ensemble
  5. Real-time anomaly detection pipeline
- 

## Section 6: Functional Models and Deployment

Duration: 8 hours | Difficulty: 

### Topics Covered:

- TensorFlow Functional API patterns
- Complex architectures (Multi-input/output, Residual, Attention)
- Data augmentation strategies (Geometric, Pixel, Advanced)
- Genetic algorithms for hyperparameter optimization
- Production deployment architectures
- Container orchestration and model serving
- Monitoring and drift detection
- A/B testing and canary deployments

### Key Concepts & Formulas:

- **Mixup:**  $\tilde{x} = \lambda x_1 + (1-\lambda)x_2$ ,  $\lambda \sim \text{Beta}(\alpha, \alpha)$
- **GA Fitness:**  $f = \text{accuracy} - \lambda \times \text{complexity}$
- **Drift Detection:** KS statistic, distribution monitoring
- **Memory Tricks:** FORMS (Flexible Operations, Reusable Modules, Shared)
- **Memory Tricks:** GASP (Geometric, Appearance, Synthetic, Probabilistic)
- **Memory Tricks:** SCALE (Serve, Containerize, Automate, Load balance, Evaluate)

## Key Skills:

- Build complex multi-branch architectures
- Implement advanced data augmentation
- Optimize models with genetic algorithms
- Deploy production-ready ML systems
- Monitor model performance and drift

## Labs:

1. Multi-modal security model with Functional API
  2. CAPTCHA solving with augmentation
  3. Genetic algorithm hyperparameter optimization
  4. Production API with monitoring
  5. Complete ML deployment pipeline
- 

## Capstone Projects

### Project 1: Enterprise Security Operations Center (SOC) ML System

**Duration:** 20 hours | **Difficulty:** ★★★★★

Build a complete ML-powered SOC system that includes:

#### 1. Data Collection Module

- Ingest logs from multiple sources (firewall, IDS, endpoint)
- Real-time stream processing
- Data normalization and storage

#### 2. Threat Detection Engine

- Multi-algorithm anomaly detection
- Known attack pattern matching
- Zero-day detection capabilities

#### 3. Incident Response Automation

- Automatic threat classification
- Severity scoring
- Playbook recommendation

#### 4. Dashboard and Alerting

- Real-time visualization
- Custom alerting rules
- Integration with SIEM

## **Deliverables:**

- Complete codebase
  - Docker deployment
  - Performance benchmarks
  - Documentation
- 

## **Project 2: Advanced Malware Analysis Platform**

**Duration:** 15 hours | **Difficulty:** ★★★★★

Create an automated malware analysis system using ML:

### **1. Static Analysis Module**

- PE header analysis
- String extraction and analysis
- Import/Export analysis
- Entropy calculation

### **2. Dynamic Analysis Integration**

- Sandbox behavior monitoring
- API call sequence analysis
- Network traffic analysis

### **3. ML Classification System**

- Family classification
- Behavior prediction
- Similarity analysis

### **4. Reporting System**

- Automated report generation
- MITRE ATT&CK mapping
- IOC extraction

## **Deliverables:**

- Analysis pipeline
  - Trained models
  - API interface
  - Sample reports
- 

## **Project 3: Insider Threat Detection System**

**Duration:** 15 hours | **Difficulty:** ★★★★☆

Develop a system to detect insider threats:

### 1. User Behavior Baseline

- Normal activity profiling
- Peer group analysis
- Temporal pattern learning

### 2. Anomaly Detection

- Unusual access patterns
- Data exfiltration detection
- Privilege escalation monitoring

### 3. Risk Scoring

- Multi-factor risk assessment
- Explainable AI for investigations
- Trend analysis

### 4. Privacy-Preserving Design

- Federated learning implementation
- Differential privacy
- Audit trail

**Deliverables:**

- Detection algorithms
- Risk scoring model
- Privacy documentation
- Deployment guide

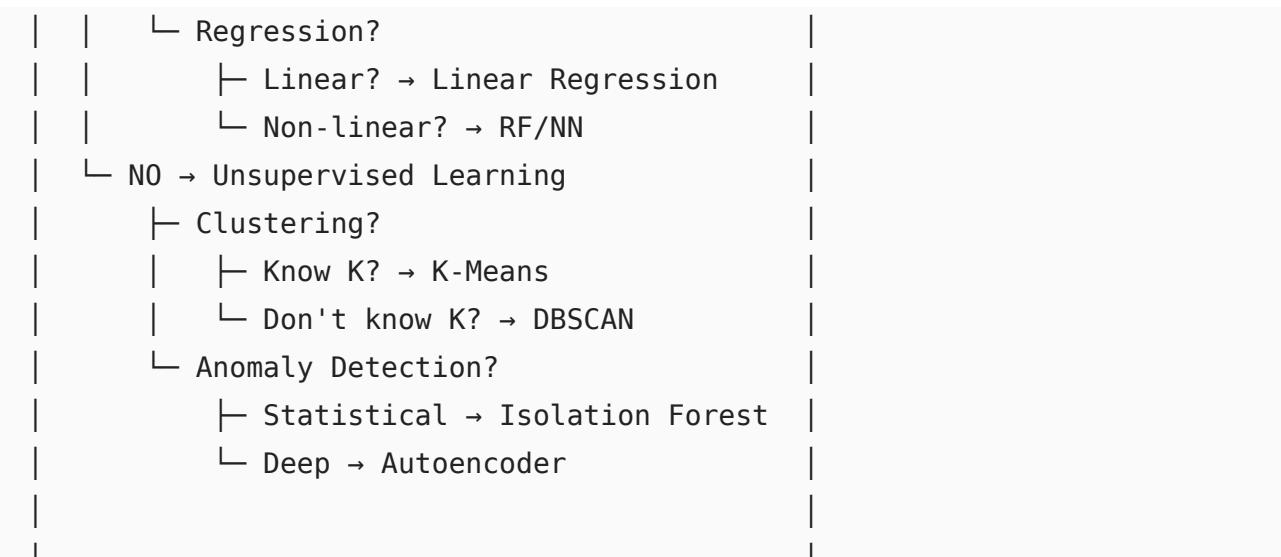


## Quick Reference Sheets

### Algorithm Selection Guide

#### ALGORITHM SELECTION TREE

- ```
| Do you have labeled data?  
|   | YES → Supervised Learning  
|   |   | Classification?  
|   |   |   | Linear? → Logistic/SVM  
|   |   |   | Non-linear? → RF/NN
```



## Performance Metrics Cheat Sheet

| Task                         | Metrics                                       | When to Use                                                    |
|------------------------------|-----------------------------------------------|----------------------------------------------------------------|
| <b>Binary Classification</b> | Accuracy, Precision, Recall, F1, ROC-AUC      | Balanced: Accuracy<br>Imbalanced: F1/ROC-AUC                   |
| <b>Multi-class</b>           | Accuracy, Macro/Micro F1, Confusion Matrix    | Macro: Equal class importance<br>Micro: Instance-weighted      |
| <b>Regression</b>            | MSE, MAE, R <sup>2</sup> , MAPE               | MSE: Penalize large errors<br>MAE: Robust to outliers          |
| <b>Clustering</b>            | Silhouette, Davies-Bouldin, Calinski-Harabasz | Silhouette: Individual point quality<br>DB: Cluster separation |
| <b>Anomaly Detection</b>     | Precision@K, ROC-AUC, Average Precision       | P@K: Top anomalies matter<br>AP: Overall ranking quality       |

## Common Hyperparameters

| Model                 | Key Parameters                                 | Typical Ranges                            |
|-----------------------|------------------------------------------------|-------------------------------------------|
| <b>Random Forest</b>  | n_estimators<br>max_depth<br>min_samples_split | 100-1000<br>3-20<br>2-20                  |
| <b>SVM</b>            | C<br>gamma (RBF)<br>kernel                     | 0.01-1000<br>0.0001-10<br>linear/rbf/poly |
| <b>Neural Network</b> | learning_rate<br>batch_size                    | 0.0001-0.1<br>16-256                      |

| Model   | Key Parameters | Typical Ranges |
|---------|----------------|----------------|
|         | dropout        | 0.2-0.5        |
| K-Means | n_clusters     | 2-20           |
|         | n_init         | 10-50          |
|         | max_iter       | 100-500        |
| DBSCAN  | eps            | 0.1-10         |
|         | min_samples    | 3-20           |

## Memory Tricks Summary

| Section   | Acronym | Meaning                                                        |
|-----------|---------|----------------------------------------------------------------|
| Section 4 | WIBA    | Weights multiply Inputs, add Bias, apply Activation            |
| Section 4 | FLEB    | Forward pass, Loss, Error backwards, update Bias & weights     |
| Section 5 | SLIP    | Slide kernel, Local interactions, Invariant, Parameter sharing |
| Section 5 | TED     | Tokens to Embedding Dimensions                                 |
| Section 5 | ELDER   | Encode, Latent space, Decode, Error, Reconstruct               |
| Section 6 | FORMS   | Flexible Operations, Reusable Modules, Shared layers           |
| Section 6 | GASP    | Geometric, Appearance, Synthetic, Probabilistic augmentation   |
| Section 6 | SCALE   | Serve, Containerize, Automate, Load balance, Evaluate          |

## Additional Resources

### Books

1. "**Hands-On Machine Learning**" by Aurélien Géron
2. "**Pattern Recognition and Machine Learning**" by Christopher Bishop
3. "**The Elements of Statistical Learning**" by Hastie, Tibshirani, and Friedman
4. "**Deep Learning**" by Ian Goodfellow, Yoshua Bengio, and Aaron Courville

### Online Courses

1. **Coursera:** Machine Learning by Andrew Ng
2. **Fast.ai:** Practical Deep Learning for Coders
3. **SANS SEC595:** Applied Data Science and Machine Learning for Cybersecurity

### Security-Specific Resources

1. **Blogs:**

- Google Security Blog
- Cisco Talos Intelligence
- FireEye Threat Research

## 2. Datasets:

- DARPA Intrusion Detection Datasets
- CTU-13 Botnet Dataset
- EMBER Malware Dataset

## 3. Tools:

- **Scikit-learn**: General ML algorithms
- **TensorFlow/PyTorch**: Deep learning
- **RAPIDS**: GPU-accelerated ML
- **MLflow**: Experiment tracking
- **Kubeflow**: ML workflows on Kubernetes

# Research Papers

## 1. Anomaly Detection:

- "Isolation Forest" (Liu et al., 2008)
- "Variational Autoencoder based Anomaly Detection" (An & Cho, 2015)

## 2. Security Applications:

- "Outside the Closed World: On Using Machine Learning for Network Intrusion Detection" (Sommer & Paxson, 2010)
- "Deep Learning for Cyber Security Intrusion Detection: Approaches, Datasets, and Comparative Study" (Ferrag et al., 2020)

# Communities

1. **Reddit**: r/MachineLearning, r/cybersecurity
2. **Discord**: MLSec Community
3. **Slack**: OWASP ML Security
4. **Twitter**: Follow [#MLSec](#) hashtag

---

# Certification Path

After completing this course, consider:

## 1. Industry Certifications:

- TensorFlow Developer Certificate
- AWS Machine Learning Specialty
- Google Cloud ML Engineer

## 2. Security-Specific:

- SANS GIAC Data Science (GDSA)
- EC-Council AI Security

## 3. Academic Path:

- Graduate certificate in ML
  - Master's in Data Science/ML
- 

## Support and Updates

- **Course Updates:** Check GitHub repository monthly
  - **Questions:** Post in course discussion forum
  - **Bug Reports:** Submit via GitHub issues
  - **Feature Requests:** Use course feedback form
- 



## What's Next?

1. **Complete all exercises:** Practice is key to mastery
  2. **Join study groups:** Collaborate with peers
  3. **Work on projects:** Apply skills to real problems
  4. **Contribute:** Share your implementations
  5. **Stay current:** ML/Security landscape evolves rapidly
- 

**Remember:** The best way to learn is by doing. Start with simple projects and gradually increase complexity. Focus on understanding concepts rather than memorizing syntax.

Good luck on your machine learning journey! 

Brought to you by Threat Route 66 <https://tr66.net>

# Quick Reference Sheets



## Section 1: Data Acquisition & Cleaning

### NumPy Essential Operations

```
# Array Creation
np.array([1,2,3])           # From list
np.zeros((3,4))             # Zeros
np.ones((2,3))              # Ones
np.arange(0,10,2)            # Sequence
np.linspace(0,1,5)           # Evenly spaced

# Array Operations
a.reshape(2,3)               # Reshape
a.T                          # Transpose
np.dot(a,b)                  # Matrix multiply
a * b                        # Element-wise
np.sum(a, axis=0)             # Sum columns
```

### SQL Optimization Checklist

1

- Use indexes on JOIN and WHERE columns
- Avoid SELECT \* - specify columns
- Use LIMIT for testing
- EXPLAIN ANALYZE for query plans
- Batch operations over loops

### Data Cleaning Pipeline

```
# Missing Data
df.isnull().sum()           # Count missing
df.fillna(method='ffill')    # Forward fill
df.dropna(thresh=0.8*len(df)) # Drop if >20% missing

# Outliers (IQR Method)
Q1, Q3 = df.quantile([0.25, 0.75])
IQR = Q3 - Q1
outliers = (df < Q1-1.5*IQR) | (df > Q3+1.5*IQR)
```

### Scaling Methods

| Method  | Formula                        | Range        | When to Use  |
|---------|--------------------------------|--------------|--------------|
| Min-Max | $(x-\min)/(max-\min)$          | [0,1]        | Known bounds |
| Z-Score | $(x-\mu)/\sigma$               | $\sim[-3,3]$ | Normal dist  |
| Robust  | $(x-\text{median})/\text{IQR}$ | Varies       | Outliers     |

---

# Descriptive Statistics

Mean:  $\mu = \Sigma x/n$   
Median: Middle value when sorted  
Mode: Most frequent value  
Variance:  $\sigma^2 = \Sigma(x-\mu)^2/n$   
Std Dev:  $\sigma = \sqrt{\text{variance}}$

# Robust Measures

```
MAD = median(|x - median(x)|)  
IQR = Q3 - Q1  
Trimmed Mean = mean(data[5%:95%])
```

# Probability Rules

```
P(A ∪ B) = P(A) + P(B) - P(A ∩ B)      # Union  
P(A ∩ B) = P(A) × P(B)                   # If independent  
P(A|B) = P(A ∩ B) / P(B)                 # Conditional
```

3

# Bayes' Theorem

```
P(A|B) = P(B|A) × P(A) / P(B)  
P(B) = Σ P(B|Ai) × P(Ai)           # Total probability
```

# Distributions Quick Reference

| Distribution | Parameters    | Use Case               |
|--------------|---------------|------------------------|
| Normal       | $\mu, \sigma$ | Natural phenomena      |
| Poisson      | $\lambda$     | Count data             |
| Exponential  | $\lambda$     | Time between events    |
| Binomial     | n, p          | Success/failure trials |

# Statistical Tests

```
# Normality  
stat, p = stats.shapiro(data)          # p > 0.05 → normal
```

```
# Comparison
t_stat, p = stats.ttest_ind(g1, g2)      # Compare means
chi2, p = stats.chi2(observed, expected) # Independence

# Effect Size
cohen_d = (mu1 - mu2) / sigma_pooled
```

---

# Algorithm Selection Matrix

| Data Type        | Problem        | Algorithm     | Key Parameter    |
|------------------|----------------|---------------|------------------|
| Tabular + Labels | Classification | Random Forest | n_estimators     |
| Tabular + Labels | Regression     | XGBoost       | max_depth        |
| No Labels        | Clustering     | DBSCAN        | eps, min_samples |
| High Dim         | Visualization  | PCA           | n_components     |
| Non-linear       | Classification | SVM RBF       | C, gamma         |

## Clustering Algorithms

```
# K-Means
kmeans = KMeans(n_clusters=k)
labels = kmeans.fit_predict(X)
inertia = kmeans.inertia_ # Within-cluster sum of squares

# DBSCAN
dbscan = DBSCAN(eps=0.5, min_samples=5)
labels = dbscan.fit_predict(X) # -1 = noise

# Optimal k (Elbow method)
inertias = [KMeans(k).fit(X).inertia_ for k in range(1,10)]
```

5

## Tree Algorithms

```
# Splitting Criteria
Gini = 1 - Σp_i^2
Entropy = -Σp_i log₂(p_i)
InfoGain = Entropy(parent) - Σ(n_i/n)×Entropy(child_i)

# Random Forest
- Bootstrap samples
- Random feature subsets
- Majority voting
```

## SVM Key Concepts

```
# Kernel Functions
Linear: K(x,y) = x·y
Polynomial: K(x,y) = (γx·y + r)^d
```

```
RBF:           K(x,y) = exp(-γ||x-y||²)

# Hyperparameters
C: Regularization (higher = less regularization)
γ: RBF width (higher = more complex boundary)
```

## Model Evaluation

```
# Classification Metrics
accuracy = (TP + TN) / total
precision = TP / (TP + FP)
recall = TP / (TP + FN)
f1 = 2 × (precision × recall) / (precision + recall)

# Cross-validation
scores = cross_val_score(model, X, y, cv=5)
```

# Neural Network Formulas

```
# Forward Pass
z = Wx + b                      # Linear combination
a = σ(z)                          # Activation

# Activation Functions
Sigmoid:   σ(x) = 1/(1+e-x)
Tanh:      tanh(x) = (ex-e-x)/(ex+e-x)
ReLU:       max(0,x)
Leaky ReLU: max(0.01x,x)
Softmax:   ex_i/Σex_j
```

7

# Loss Functions

| Problem      | Loss Function             | Formula                                   |
|--------------|---------------------------|-------------------------------------------|
| Regression   | MSE                       | (1/n)Σ(y-ŷ) <sup>2</sup>                  |
| Regression   | MAE                       | (1/n)Σ                                    |
| Binary Class | Binary Cross-Entropy      | [-y·log(ŷ)+(1-y)·log(1-ŷ)]                |
| Multi-class  | Categorical Cross-Entropy | -ΣΣy <sub>ij</sub> ·log(ŷ <sub>ij</sub> ) |

# Gradient Descent

```
θ_new = θ_old - α × ∇L(θ)

# Learning Rate Schedules
Exponential: α_t = α0 × decayt
Step:        α_t = α0 × dropfloor(t/epochs_drop)
Cosine:      α_t = α0 × 0.5 × (1 + cos(πt/T))
```

# Regularization Techniques

```
# L2 Regularization
loss = mse + λ × Σw2

# Dropout
training: output = input × mask, mask ~ Bernoulli(p)
inference: output = input × p

# Batch Normalization
```

```
x_norm = (x - μ_batch) / √(σ²_batch + ε)  
y = γ × x_norm + β
```

## Memory Tricks

- **WIBA:** Weights, Inputs, Bias, Activation
  - **FLEB:** Forward, Loss, Error backward, Bias/weight update
-

## Convolution Formulas

```
# 1D Convolution  
(f * g)[n] = Σ f[m] × g[n-m]  
  
# 2D Convolution  
(f * g)[i,j] = ΣΣ f[m,n] × g[i-m,j-n]  
  
# Output Size  
out = ⌊(in + 2×pad - kernel) / stride⌋ + 1
```

## CNN Architecture Guidelines

```
# Typical Architecture  
Input → Conv → ReLU → Pool → Conv → ReLU → Pool → Flatten → Dense → Output  
  
# Filter Progression  
32 → 64 → 128 → 256 (double each layer)  
  
# Kernel Sizes  
First layers: 5×5 or 7×7 (large receptive field)  
Deep layers: 3×3 (computational efficiency)
```

9

## Autoencoder Loss Functions

```
# Reconstruction Loss  
MSE: L = (1/n)Σ||x - ū||2  
MAE: L = (1/n)Σ|x - ū|  
Binary: L = -Σ[x·log(ŷ) + (1-x)·log(1-ŷ)]  
  
# Variational (VAE)  
L = Reconstruction + β×KL_divergence  
KL = -0.5 × Σ(1 + log(σ2) - μ2 - σ2)
```

## Anomaly Detection with AE

```
# Training (normal data only)  
ae.fit(X_normal, X_normal)  
  
# Detection  
reconstruction = ae.predict(X_test)  
error = mse(X_test, reconstruction)
```

```
anomaly = error > threshold

# Threshold Selection
threshold = μ_error + k×σ_error      # k ∈ [2,3]
threshold = percentile(errors, 95)    # Top 5% as anomalies
```

## Memory Tricks

- **SLIP**: Slide, Local, Invariant, Parameter sharing
  - **TED**: Tokens to Embedding Dimensions
  - **ELDER**: Encode, Latent, Decode, Error, Reconstruct
-

# Functional API Patterns

```
# Multi-Input
input1 = Input(shape1)
input2 = Input(shape2)
merged = concatenate([branch1, branch2])

# Multi-Output
shared = Dense(128)(input)
out1 = Dense(n1, activation='softmax')(shared)
out2 = Dense(n2, activation='sigmoid')(shared)

# Residual Connection
y = Conv2D(64)(x)
y = BatchNormalization()(y)
out = Add()([x, y]) # Skip connection
```

# Data Augmentation

```
# Image Augmentation
ImageDataGenerator(
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    horizontal_flip=True,
    zoom_range=0.2
)

# Custom Augmentation
def mixup(x1, x2, y1, y2, alpha=0.2):
    lam = np.random.beta(alpha, alpha)
    x = lam * x1 + (1 - lam) * x2
    y = lam * y1 + (1 - lam) * y2
    return x, y
```

11

# Genetic Algorithm Components

```
# Fitness Function
fitness = accuracy - λ×complexity

# Selection (Tournament)
winner = max(random.sample(population, k))

# Crossover (Single-point)
child = parent1[:point] + parent2[point:]
```

```
# Mutation
if random() < p_mutate:
    gene = gene + N(0, σ²)
```

## Deployment Checklist

### Model:

- ✓ Versioned (model\_v1.2.3)
- ✓ Serialized (SavedModel, ONNX)
- ✓ Optimized (quantization, pruning)

### API:

- ✓ Input validation
- ✓ Rate limiting
- ✓ Authentication
- ✓ Error handling

### Container:

- ✓ Dockerfile optimized
- ✓ Security scanned
- ✓ Resource limits set

### Monitoring:

- ✓ Latency (p50, p95, p99)
- ✓ Throughput (req/sec)
- ✓ Error rate
- ✓ Model drift

12

## Production Metrics

```
# Latency Monitoring
latency_p95 = np.percentile(latencies, 95)
alert if latency_p95 > 100ms

# Drift Detection
drift_score = KS_statistic(train_dist, prod_dist)
alert if drift_score > 0.1

# Performance Monitoring
accuracy_window = accuracy(last_1000_predictions)
alert if accuracy_window < baseline - 0.05
```

## Memory Tricks

- **FORMS:** Flexible Operations, Reusable Modules, Shared

- **GASP**: Geometric, Appearance, Synthetic, Probabilistic
  - **SCALE**: Serve, Containerize, Automate, Load balance, Evaluate
-

# Data Pipeline

1. **Validate early:** Check data quality at ingestion
2. **Version everything:** Data, code, models
3. **Automate cleaning:** Consistent preprocessing
4. **Monitor drift:** Track distribution changes

# Model Development

1. **Start simple:** Baseline → Complex
2. **Validate properly:** Temporal splits for time series
3. **Ensemble when possible:** Reduces overfitting
4. **Document assumptions:** Future-proof your work

# Production Deployment

1. **A/B test:** Compare models safely
2. **Canary deploy:** Gradual rollout
3. **Monitor everything:** Can't improve what you don't measure
4. **Plan rollback:** Always have escape route

# Debugging Checklist

- Data issues? → Check preprocessing pipeline
  - Overfitting? → Add regularization, get more data
  - Underfitting? → Increase capacity, engineer features
  - Slow training? → Batch size, learning rate, architecture
  - Poor generalization? → Cross-validation, data leakage
-

## Model Training

```
# Scikit-learn
model.fit(X_train, y_train)
predictions = model.predict(X_test)
score = model.score(X_test, y_test)

# TensorFlow/Keras
model.compile(optimizer='adam', loss='mse', metrics=['mae'])
history = model.fit(X_train, y_train, validation_split=0.2, epochs=50)
loss, mae = model.evaluate(X_test, y_test)
```

## Performance Optimization

```
# Vectorization
# Bad: for i in range(len(x)): y[i] = x[i] * 2
# Good: y = x * 2

# Batch Processing
# Bad: for sample in data: predict(sample)
# Good: predictions = predict_batch(data)

# GPU Acceleration
with tf.device('/GPU:0'):
    model.fit(X, y)
```

## Algorithm Selection

Tabular + Small → Traditional ML (RF, XGBoost)

Tabular + Large → Neural Networks

Images → CNN

Sequences → RNN/LSTM/Transformer

Unlabeled → Clustering, Autoencoders

Real-time → Optimize for latency

## Metric Selection

Balanced Classes → Accuracy

Imbalanced → F1, AUC-ROC

Cost-sensitive → Custom metric

Ranking → MAP, NDCG

Regression → RMSE (outlier sensitive), MAE (robust)

## Architecture Selection

Simple Patterns → Shallow networks

Complex Patterns → Deep networks

Multiple Inputs → Functional API

Transfer Learning → Pre-trained + Fine-tune

Limited Data → Data augmentation + Regularization

---

# Algorithm Quick Reference

## Data Preprocessing

```
# Standard imports
import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.model_selection import train_test_split

# Load and split data
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42, stratify=y
)

# Scaling
scaler = StandardScaler() # or MinMaxScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test) # Only transform!

# Handle missing data
df.fillna(df.mean(), inplace=True) # Numerical
df.fillna(df.mode()[0], inplace=True) # Categorical
```

17

---

# Clustering Algorithms

## K-Means

```
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score

# Find optimal K
silhouette_scores = []
for k in range(2, 10):
    kmeans = KMeans(n_clusters=k, random_state=42)
    labels = kmeans.fit_predict(X_scaled)
    silhouette_scores.append(silhouette_score(X_scaled, labels))

# Train model
kmeans = KMeans(n_clusters=optimal_k, n_init=10, random_state=42)
clusters = kmeans.fit_predict(X_scaled)

# Analyze clusters
```

```
for i in range(optimal_k):
    cluster_data = X[clusters == i]
    print(f"Cluster {i}: {len(cluster_data)} samples")
    print(f"Centroid: {kmeans.cluster_centers_[i]}")
```

## DBSCAN

```
from sklearn.cluster import DBSCAN
from sklearn.neighbors import NearestNeighbors

# Find optimal epsilon
neighbors = NearestNeighbors(n_neighbors=5)
neighbors_fit = neighbors.fit(X_scaled)
distances, indices = neighbors_fit.kneighbors(X_scaled)
distances = np.sort(distances[:, 4], axis=0)
# Plot distances and look for elbow

# Apply DBSCAN
dbSCAN = DBSCAN(eps=0.5, min_samples=5)
clusters = dbSCAN.fit_predict(X_scaled)
n_clusters = len(set(clusters)) - (1 if -1 in clusters else 0)
n_noise = list(clusters).count(-1)
```

## Decision Tree

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import classification_report

dt = DecisionTreeClassifier(
    max_depth=5,
    min_samples_split=20,
    min_samples_leaf=10,
    random_state=42
)
dt.fit(X_train, y_train)

# Feature importance
importance = pd.DataFrame({
    'feature': feature_names,
    'importance': dt.feature_importances_
}).sort_values('importance', ascending=False)

# Predictions
y_pred = dt.predict(X_test)
print(classification_report(y_test, y_pred))
```

19

## Random Forest

```
from sklearn.ensemble import RandomForestClassifier

rf = RandomForestClassifier(
    n_estimators=100,
    max_depth=10,
    min_samples_split=5,
    oob_score=True, # Out-of-bag score
    random_state=42,
    n_jobs=-1 # Use all cores
)
rf.fit(X_train, y_train)

print(f"OOB Score: {rf.oob_score_:.3f}")

# Feature importance with permutation
from sklearn.inspection import permutation_importance
perm_importance = permutation_importance(rf, X_test, y_test, n_repeats=10)
```

## Support Vector Machine

```
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV

# Hyperparameter tuning
param_grid = {
    'C': [0.1, 1, 10, 100],
    'gamma': ['scale', 'auto', 0.001, 0.01, 0.1],
    'kernel': ['rbf', 'linear']
}

svm = SVC(probability=True, random_state=42)
grid_search = GridSearchCV(svm, param_grid, cv=5, n_jobs=-1)
grid_search.fit(X_train_scaled, y_train)

best_svm = grid_search.best_estimator_
```

---

## Basic Dense Network

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

model = keras.Sequential([
    layers.Dense(128, activation='relu', input_shape=(n_features,)),
    layers.BatchNormalization(),
    layers.Dropout(0.3),

    layers.Dense(64, activation='relu'),
    layers.BatchNormalization(),
    layers.Dropout(0.2),

    layers.Dense(32, activation='relu'),
    layers.Dense(n_classes, activation='softmax')
])

model.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)

# Callbacks
callbacks = [
    keras.callbacks.EarlyStopping(patience=10, restore_best_weights=True),
    keras.callbacks.ReduceLROnPlateau(patience=5, factor=0.5),
    keras.callbacks.ModelCheckpoint('best_model.h5', save_best_only=True)
]

history = model.fit(
    X_train, y_train,
    validation_split=0.2,
    epochs=100,
    batch_size=32,
    callbacks=callbacks
)
```

# For sequences (1D CNN)

```
model = keras.Sequential([
    layers.Embedding(vocab_size, embedding_dim, input_length=max_length),
    layers.Conv1D(128, 5, activation='relu'),
    layers.GlobalMaxPooling1D(),
    layers.Dense(64, activation='relu'),
    layers.Dropout(0.5),
    layers.Dense(1, activation='sigmoid')
])
```

```
<div style="page-break-after: always;"></div>
### Autoencoder for Anomaly Detection
```python
# Encoder
encoder_input = layers.Input(shape=(n_features,))
encoded = layers.Dense(64, activation='relu')(encoder_input)
encoded = layers.Dense(32, activation='relu')(encoded)
encoded = layers.Dense(16, activation='relu')(encoded)

# Decoder
decoded = layers.Dense(32, activation='relu')(encoded)
decoded = layers.Dense(64, activation='relu')(decoded)
decoded = layers.Dense(n_features, activation='sigmoid')(decoded)

# Autoencoder
autoencoder = keras.Model(encoder_input, decoded)
encoder = keras.Model(encoder_input, encoded)

autoencoder.compile(optimizer='adam', loss='mse')

# Train on normal data only
autoencoder.fit(X_normal, X_normal, epochs=50, batch_size=32)

# Detect anomalies
reconstructions = autoencoder.predict(X_test)
mse = np.mean((X_test - reconstructions)**2, axis=1)
threshold = np.percentile(mse, 95)
anomalies = mse > threshold
```

# Classification Metrics

```
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score, f1_score,
    roc_auc_score, confusion_matrix, classification_report
)

# Basic metrics
accuracy = accuracy_score(y_true, y_pred)
precision = precision_score(y_true, y_pred, average='weighted')
recall = recall_score(y_true, y_pred, average='weighted')
f1 = f1_score(y_true, y_pred, average='weighted')

# For probability predictions
auc = roc_auc_score(y_true, y_proba, multi_class='ovr')

# Confusion matrix
cm = confusion_matrix(y_true, y_pred)
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.title('Confusion Matrix')
plt.ylabel('True Label')
plt.xlabel('Predicted Label')

# Detailed report
print(classification_report(y_true, y_pred))
```

23

# Cross-Validation

```
from sklearn.model_selection import cross_val_score, StratifiedKFold

# Simple cross-validation
scores = cross_val_score(model, X, y, cv=5, scoring='accuracy')
print(f"CV Score: {scores.mean():.3f} (+/- {scores.std()*2:.3f})")

# Stratified K-Fold for imbalanced data
skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
scores = cross_val_score(model, X, y, cv=skf, scoring='f1_weighted')
```

# Model Serialization

```
import joblib
import pickle

# Scikit-learn models
joblib.dump(model, 'model.pkl')
model = joblib.load('model.pkl')

# Include preprocessing
pipeline = {
    'scaler': scaler,
    'model': model,
    'features': feature_names,
    'version': '1.0'
}
joblib.dump(pipeline, 'pipeline.pkl')

# TensorFlow/Keras
model.save('model.h5') # HDF5 format
model.save('model_dir') # SavedModel format
loaded_model = keras.models.load_model('model.h5')
```

24

# Simple Flask API

```
from flask import Flask, request, jsonify
import numpy as np

app = Flask(__name__)
model = joblib.load('model.pkl')
scaler = joblib.load('scaler.pkl')

@app.route('/predict', methods=['POST'])
def predict():
    try:
        # Get data
        data = request.json['features']

        # Preprocess
        features = np.array(data).reshape(1, -1)
        features_scaled = scaler.transform(features)

        # Predict
        prediction = model.predict(features_scaled)[0]
        probability = model.predict_proba(features_scaled)[0].max()

    return jsonify({
```

```

        'prediction': int(prediction),
        'confidence': float(probability),
        'status': 'success'
    })

except Exception as e:
    return jsonify({'error': str(e), 'status': 'error'}), 400

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)

```

## Docker Deployment

```

FROM python:3.9-slim

WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY . .

EXPOSE 5000

CMD ["python", "app.py"]

```

## Issue: Overfitting

```
# Solutions:  
# 1. More data or data augmentation  
# 2. Regularization  
model.add(layers.Dense(64,  
kernel_regularizer=keras.regularizers.l2(0.01)))  
# 3. Dropout  
model.add(layers.Dropout(0.5))  
# 4. Early stopping  
# 5. Simpler model architecture
```

## Issue: Class Imbalance

```
# Solutions:  
# 1. Class weights  
from sklearn.utils.class_weight import compute_class_weight  
class_weights = compute_class_weight(  
    'balanced', classes=np.unique(y_train), y=y_train  
)  
  
# 2. SMOTE  
from imblearn.over_sampling import SMOTE  
smote = SMOTE(random_state=42)  
X_resampled, y_resampled = smote.fit_resample(X_train, y_train)  
  
# 3. Different metrics (not accuracy)  
# Use: precision, recall, F1, AUC-ROC
```

26

## Issue: Slow Training

```
# Solutions:  
# 1. Reduce batch size (for memory)  
# 2. Use GPU  
# 3. Reduce model complexity  
# 4. Use transfer learning  
# 5. Implement data generators  
  
class DataGenerator(keras.utils.Sequence):  
    def __init__(self, X, y, batch_size=32):  
        self.X = X  
        self.y = y  
        self.batch_size = batch_size  
  
    def __len__(self):  
        return len(self.X) // self.batch_size
```

```
def __getitem__(self, idx):
    batch_X = self.X[idx * self.batch_size:(idx + 1) *
self.batch_size]
    batch_y = self.y[idx * self.batch_size:(idx + 1) *
self.batch_size]
    return batch_X, batch_y
```

---

# Network Anomaly Detection Pipeline

```
# 1. Feature extraction
features = ['packet_size', 'duration', 'protocol', 'flags', 'bytes_sent']

# 2. Time-window aggregation
window_features = df.groupby(['src_ip', 'time_window']).agg({
    'packet_size': ['mean', 'std', 'max'],
    'duration': ['sum', 'mean'],
    'bytes_sent': ['sum', 'mean']
}).reset_index()

# 3. Anomaly detection
iso_forest = IsolationForest(contamination=0.01, random_state=42)
anomalies = iso_forest.fit_predict(window_features)

# 4. Alert generation
alerts = window_features[anomalies == -1]
```

# Malware Detection Pattern

```
# 1. Feature extraction from PE files
pe_features = {
    'file_size': file_size,
    'entropy': calculate_entropy(file_bytes),
    'num_sections': len(pe.sections),
    'imports': len(pe.imports),
    'exports': len(pe.exports)
}

# 2. Static analysis features
strings = extract_strings(file_path)
api_calls = extract_api_calls(pe)

# 3. Vectorization
from sklearn.feature_extraction.text import TfidfVectorizer
vectorizer = TfidfVectorizer(max_features=1000)
string_features = vectorizer.fit_transform(strings)

# 4. Ensemble prediction
predictions = []
predictions.append(static_model.predict(pe_features))
predictions.append(string_model.predict(string_features))
predictions.append(api_model.predict(api_features))

final_prediction = np.array(predictions).mean(axis=0) > 0.5
```



## Before Training

- Explore data thoroughly (distributions, correlations)
- Handle missing values appropriately
- Scale/normalize features
- Split data properly (train/val/test)
- Check class balance
- Set random seeds for reproducibility

## During Training

- Monitor training and validation metrics
- Use callbacks (early stopping, checkpointing)
- Save best model, not last model
- Log experiments and hyperparameters
- Visualize learning curves

## After Training

- Evaluate on test set (only once!)
- Check for overfitting
- Analyze errors and edge cases
- Test on production-like data
- Document model limitations
- Set up monitoring

30

## In Production

- Version models and data
  - Monitor prediction distribution
  - Track latency and throughput
  - Implement fallback mechanisms
  - Regular retraining schedule
  - A/B testing for updates
-

# Documentation

- **Scikit-learn:** <https://scikit-learn.org/stable/>
- **TensorFlow:** <https://www.tensorflow.org/>
- **Pandas:** <https://pandas.pydata.org/>
- **NumPy:** <https://numpy.org/>

# Security ML Resources

- **MITRE ATT&CK:** <https://attack.mitre.org/>
- **STIX/TAXII:** <https://oasis-open.github.io/cti-documentation/>
- **Cyber Threat Intelligence:** <https://www.cisa.gov/>

# Tools

- **Jupyter:** <https://jupyter.org/>
- **MLflow:** <https://mlflow.org/>
- **DVC:** <https://dvc.org/>
- **Weights & Biases:** <https://wandb.ai/>

**Remember:** This is a reference guide. Always understand the algorithms and their assumptions before applying them to security problems. Test thoroughly and validate results with domain experts.

# Machine Learning Training Documentation

## Section 1: Data Acquisition, Cleaning, and Manipulation

### Overview

This section introduces the fundamental concepts and practical skills needed for data acquisition and manipulation in machine learning projects. We'll explore various data sources, understand data types and structures, and learn essential cleaning techniques that form the foundation of any successful ML project.

### Learning Objectives

By the end of this section, you will be able to:

- Understand and use Python data structures effectively
- Write efficient SQL queries for data retrieval
- Work with NoSQL databases for unstructured data
- Implement web scraping techniques responsibly
- Apply appropriate data cleaning strategies
- Create effective visualizations with matplotlib
- Prepare data for machine learning algorithms

# Core Concepts

## What is NumPy?

NumPy (Numerical Python) is the fundamental package for scientific computing in Python. It provides:

- **N-dimensional arrays:** Efficient storage and operations on homogeneous data
- **Vectorization:** Perform operations on entire arrays without writing loops
- **Broadcasting:** Perform operations on arrays of different shapes
- **Mathematical functions:** Optimized C implementations for speed

## Arrays vs Lists: Key Differences

| Aspect           | Python List                   | NumPy Array                 |
|------------------|-------------------------------|-----------------------------|
| Type Flexibility | Can hold mixed types          | Homogeneous (same type)     |
| Memory Usage     | Higher overhead               | Compact, contiguous memory  |
| Performance      | Slower for math operations    | 10-100x faster              |
| Operations       | Element-by-element with loops | Vectorized operations       |
| Functionality    | Basic operations              | Rich mathematical functions |

33

## Memory Representation

```
Python List: [1, 2, 3]
└─ Pointer to PyObject 1
└─ Pointer to PyObject 2
└─ Pointer to PyObject 3

NumPy Array: array([1, 2, 3])
└─ Contiguous memory block: [1|2|3]
```

## Essential NumPy Functions

### Array Creation Functions

| Function   | Purpose                   | Example             |
|------------|---------------------------|---------------------|
| np.array() | Create from existing data | np.array([1, 2, 3]) |
| np.zeros() | Initialize with zeros     | np.zeros((3, 4))    |
| np.ones()  | Initialize with ones      | np.ones((2, 3))     |

| Function          | Purpose                    | Example               |
|-------------------|----------------------------|-----------------------|
| np.arange()       | Create sequence            | np.arange(0, 10, 2)   |
| np.linspace()     | Evenly spaced values       | np.linspace(0, 1, 5)  |
| np.random.randn() | Random normal distribution | np.random.randn(3, 3) |

## Shape Manipulation Functions

| Function      | Purpose            | Memory Trick      |
|---------------|--------------------|-------------------|
| reshape()     | Change dimensions  | "Mold the clay"   |
| flatten()     | Make 1D copy       | "Pancake maker"   |
| transpose()   | Swap axes          | "Flip the matrix" |
| squeeze()     | Remove size-1 dims | "Squeeze out air" |
| expand_dims() | Add new axis       | "Add a dimension" |

## Python Data Structures for ML

### When to Use Each Structure

34

Dictionary → Key-value pairs, feature mappings

- |— Use case: Store model hyperparameters
- |— Example: {'learning\_rate': 0.01, 'epochs': 100}
- |— O(1) average lookup time

List → Ordered, mutable sequences

- |— Use case: Collect data samples
- |— Example: [sample1, sample2, sample3]
- |— O(n) search, O(1) append

Tuple → Immutable sequences

- |— Use case: Fixed configurations, coordinates
- |— Example: (width, height, channels)
- |— Memory efficient, hashable

Set → Unique elements only

- |— Use case: Track unique values, remove duplicates
- |— Example: {label1, label2, label3}
- |— O(1) average membership testing

## Code Implementation

Now let's see these concepts in practice:

```
import numpy as np
import pandas as pd

# Demonstrating array vs list performance
import time

# List operations
python_list = list(range(1000000))
start = time.time()
squared_list = [x**2 for x in python_list]
list_time = time.time() - start

# NumPy operations
numpy_array = np.array(python_list)
start = time.time()
squared_array = numpy_array**2
numpy_time = time.time() - start

print(f"List comprehension time: {list_time:.4f} seconds")
print(f"NumPy vectorization time: {numpy_time:.4f} seconds")
print(f"NumPy is {list_time/numpy_time:.1f}x faster")

# Practical example: Security log analysis
# Dictionary for mapping threat levels
threat_levels = {
    'low': 1,
    'medium': 2,
    'high': 3,
    'critical': 4
}

# Using appropriate data structures
security_events = [] # List for ordered events
unique_ips = set() # Set for unique IP addresses
ip_counts = {} # Dictionary for counting occurrences

# Simulate processing logs
sample_logs = [
    ('192.168.1.100', 'medium'),
    ('10.0.0.5', 'high'),
    ('192.168.1.100', 'low'),
    ('172.16.0.1', 'critical')
]

for ip, threat in sample_logs:
    security_events.append((ip, threat, threat_levels[threat]))
    unique_ips.add(ip)
```

```
ip_counts[ip] = ip_counts.get(ip, 0) + 1

print(f"\nTotal events: {len(security_events)}")
print(f"Unique IPs: {len(unique_ips)}")
print(f"IP frequencies: {ip_counts}")
```

---

# SQL Fundamentals

## What is SQL?

SQL (Structured Query Language) is a standardized language for managing relational databases. In data science, it's essential for:

- **Data extraction:** Retrieving specific subsets of data
- **Data aggregation:** Computing statistics directly in the database
- **Data joining:** Combining information from multiple tables
- **Data filtering:** Selecting relevant records efficiently

## SQL Command Categories

| Category                  | Purpose             | Key Commands           |
|---------------------------|---------------------|------------------------|
| DDL (Data Definition)     | Define structure    | CREATE, ALTER, DROP    |
| DML (Data Manipulation)   | Modify data         | INSERT, UPDATE, DELETE |
| DQL (Data Query)          | Retrieve data       | SELECT                 |
| DCL (Data Control)        | Manage permissions  | GRANT, REVOKE          |
| TCL (Transaction Control) | Manage transactions | COMMIT, ROLLBACK       |

37

## SQL Functions by Type

### Basic Aggregation Functions

Functions that operate on groups of rows to produce a single result:

| Function | Purpose    | Example                   | When to Use            |
|----------|------------|---------------------------|------------------------|
| COUNT()  | Count rows | COUNT(*) or COUNT(column) | Get totals             |
| SUM()    | Add values | SUM(amount)               | Total calculations     |
| AVG()    | Average    | AVG(response_time)        | Find typical values    |
| MIN()    | Minimum    | MIN(timestamp)            | Find earliest/smallest |
| MAX()    | Maximum    | MAX(severity_score)       | Find latest/largest    |

### Statistical Aggregate Functions

| Function           | Purpose            | Example               |
|--------------------|--------------------|-----------------------|
| STDDEV() / STDEV() | Standard deviation | STDDEV(response_time) |

| Function           | Purpose             | Example                   |
|--------------------|---------------------|---------------------------|
| VARIANCE() / VAR() | Variance            | VARIANCE(packet_size)     |
| STDDEV_POP()       | Population std dev  | STDDEV_POP(cpu_usage)     |
| STDDEV_SAMP()      | Sample std dev      | STDDEV_SAMP(memory_usage) |
| VAR_POP()          | Population variance | VAR_POP(connection_count) |
| VAR_SAMP()         | Sample variance     | VAR_SAMP(error_rate)      |

## Advanced Aggregate Functions

| Function          | Database   | Purpose                    | Example                   |
|-------------------|------------|----------------------------|---------------------------|
| GROUP_CONCAT()    | MySQL      | Concatenate values         | GROUP_CONCAT(ip_address)  |
| STRING_AGG()      | PostgreSQL | Concatenate with delimiter | STRING_AGG(username, ',') |
| LISTAGG()         | Oracle     | List aggregation           | LISTAGG(event_type, ';' ) |
| ARRAY_AGG()       | PostgreSQL | Create array               | ARRAY_AGG(port_number)    |
| JSON_AGG()        | PostgreSQL | Create JSON array          | JSON_AGG(log_entry)       |
| PERCENTILE_CONT() | SQL:2003   | Continuous percentile      | PERCENTILE_CONT(0.95)     |
| PERCENTILE_DISC() | SQL:2003   | Discrete percentile        | PERCENTILE_DISC(0.75)     |

38

## Window Function Aggregates

Perform calculations across a set of rows while preserving individual rows:

```
-- Syntax pattern
FUNCTION() OVER (PARTITION BY column ORDER BY column)

-- Common window functions
ROW_NUMBER()      -- Sequential numbering
RANK()            -- Ranking with gaps
DENSE_RANK()      -- Ranking without gaps
LAG()             -- Previous row value
LEAD()            -- Next row value
FIRST_VALUE()     -- First value in window
LAST_VALUE()      -- Last value in window
```

## Important Notes on Aggregation

- NULL Handling:** Most aggregate functions ignore NULL values except COUNT(\*)
- GROUP BY Rule:** When using aggregates, all non-aggregate columns must be in GROUP BY
- HAVING vs WHERE:** Use HAVING to filter aggregated results, WHERE for row-level filtering
- DISTINCT:** Can be used within aggregates: COUNT(DISTINCT user\_id)

## SQL Best Practices for Data Science

- Always use parameterized queries** to prevent SQL injection
- Index columns** used in WHERE, JOIN, and ORDER BY clauses
- Limit result sets** during exploration with LIMIT
- Use CTEs** (Common Table Expressions) for complex queries
- Avoid SELECT \*** in production - specify columns needed

## Query Optimization Tips

| Problem             | Solution           | Example                                     |
|---------------------|--------------------|---|
| Slow joins          | Index foreign keys | CREATE INDEX idx_user_id ON events(user_id) |
| Large results       | Add filters early  | WHERE date > '2024-01-01' before JOIN       |
| Complex logic       | Use CTEs           | WITH filtered AS (SELECT...)                |
| Repeated subqueries | Create temp tables | CREATE TEMP TABLE summary AS...             |

39

## SQL Aggregation Functions - Comprehensive Guide

### Basic Aggregation Functions

Functions that operate on groups of rows to produce a single result:

| Function | Purpose    | Example                   | Notes  |
|----------|------------|---------------------------|--|
| COUNT()  | Count rows | COUNT(*) or COUNT(column) | COUNT(*) includes NULLs,<br>COUNT(column) excludes |
| SUM()    | Add values | SUM(amount)               | Returns NULL if all values are NULL                |
| AVG()    | Average    | AVG(response_time)        | Ignores NULL values                                |
| MIN()    | Minimum    | MIN(timestamp)            | Works with dates, strings, numbers                 |

| Function       | Purpose     | Example                                 | Notes                              |
|----------------|-------------|---|------------------------------------|
| MAX()          | Maximum     | MAX(severity_score)                     | Works with dates, strings, numbers |
| GROUP_CONCAT() | Concatenate | GROUP_CONCAT(ip_address SEPARATOR ', ') | MySQL/SQLite specific              |
| STRING_AGG()   | Concatenate | STRING_AGG(ip_address, ', ')            | PostgreSQL/SQL Server              |

## Statistical Aggregation Functions

| Function          | Purpose            | Example   | Database Support        |
|-------------------|--------------------|---|-------------------------|
| STDDEV()          | Standard deviation | STDDEV(response_time)                                 | Most databases          |
| VARIANCE()        | Variance           | VARIANCE(packet_size)                                 | Most databases          |
| MEDIAN()          | Median value       | MEDIAN(salary)  | Oracle, PostgreSQL 9.4+ |
| MODE()            | Most frequent      | MODE() WITHIN GROUP (ORDER BY status)                 | PostgreSQL              |
| PERCENTILE_CONT() | Percentile         | PERCENTILE_CONT(0.95) WITHIN GROUP (ORDER BY latency) | PostgreSQL, SQL Server  |

40

## Advanced Aggregation Patterns

### 1. Conditional Aggregation

```
-- Count events by severity
SELECT
    COUNT(CASE WHEN severity = 'HIGH' THEN 1 END) as high_count,
    COUNT(CASE WHEN severity = 'MEDIUM' THEN 1 END) as medium_count,
    COUNT(CASE WHEN severity = 'LOW' THEN 1 END) as low_count,
    COUNT(*) as total_count
FROM security_events;

-- Sum with conditions
SELECT
    SUM(CASE WHEN protocol = 'HTTP' THEN bytes_transferred ELSE 0 END) as http_bytes,
    SUM(CASE WHEN protocol = 'HTTPS' THEN bytes_transferred ELSE 0 END) as https_bytes;
```

```
https_bytes  
FROM network_traffic;
```

## 2. Multiple Aggregations

```
-- Combined statistics  
SELECT  
    category,  
    COUNT(*) AS event_count,  
    AVG(severity) AS avg_severity,  
    MIN(timestamp) AS first_seen,  
    MAX(timestamp) AS last_seen,  
    COUNT(DISTINCT source_ip) AS unique_sources  
FROM security_events  
GROUP BY category;
```

## 3. Nested Aggregations

```
-- Average of daily counts  
WITH daily_counts AS (  
    SELECT  
        DATE(timestamp) AS event_date,  
        COUNT(*) AS daily_count  
    FROM security_events  
    GROUP BY DATE(timestamp)  
)  
SELECT  
    AVG(daily_count) AS avg_daily_events,  
    MAX(daily_count) AS max_daily_events,  
    MIN(daily_count) AS min_daily_events  
FROM daily_counts;
```

41

## Window Functions with Aggregation

Perform calculations across a set of rows while preserving individual rows:

```
-- Running totals  
SELECT  
    timestamp,  
    source_ip,  
    COUNT(*) OVER (PARTITION BY source_ip ORDER BY timestamp) AS  
running_count,  
    SUM(bytes) OVER (PARTITION BY source_ip ORDER BY timestamp) AS  
running_bytes  
FROM network_traffic;  
  
-- Ranking within groups
```

```

SELECT
    category,
    source_ip,
    event_count,
    RANK() OVER (PARTITION BY category ORDER BY event_count DESC) as
rank_in_category
FROM (
    SELECT category, source_ip, COUNT(*) as event_count
    FROM security_events
    GROUP BY category, source_ip
) t;

-- Moving averages
SELECT
    timestamp,
    response_time,
    AVG(response_time) OVER (
        ORDER BY timestamp
        ROWS BETWEEN 4 PRECEDING AND CURRENT ROW
    ) as moving_avg_5
FROM api_logs;

```

## HAVING Clause with Aggregations

Filter groups after aggregation:

42

```

-- Find IPs with suspicious activity
SELECT
    source_ip,
    COUNT(*) as request_count,
    COUNT(DISTINCT destination_port) as unique_ports,
    AVG(bytes_transferred) as avg_bytes
FROM network_traffic
WHERE timestamp > NOW() - INTERVAL '1 hour'
GROUP BY source_ip
HAVING
    COUNT(*) > 1000
    OR COUNT(DISTINCT destination_port) > 100
    OR AVG(bytes_transferred) > 1000000;

```

## Practical Security Analytics Examples

### 1. DDoS Detection Query

```

-- Detect potential DDoS sources
SELECT
    source_ip,
    COUNT(*) as request_count,

```

```

        COUNT(DISTINCT destination_ip) as targets,
        AVG(CASE WHEN response_code = 200 THEN 1 ELSE 0 END) as success_rate,
        MIN(timestamp) as first_seen,
        MAX(timestamp) as last_seen,
        (MAX(timestamp) - MIN(timestamp)) as attack_duration
    FROM web_logs
    WHERE timestamp > NOW() - INTERVAL '10 minutes'
    GROUP BY source_ip
    HAVING COUNT(*) > 1000
    ORDER BY request_count DESC;

```

## 2. Port Scan Detection

```

-- Identify potential port scanners
WITH port_scans AS (
    SELECT
        source_ip,
        destination_ip,
        COUNT(DISTINCT destination_port) as ports_scanned,
        COUNT(*) as total_attempts,
        COUNT(DISTINCT DATE_TRUNC('second', timestamp)) as
scan_duration_seconds
    FROM network_traffic
    WHERE timestamp > NOW() - INTERVAL '1 hour'
    GROUP BY source_ip, destination_ip
)
SELECT
    source_ip,
    COUNT(DISTINCT destination_ip) as targets,
    SUM(ports_scanned) as total_ports_scanned,
    AVG(ports_scanned) as avg_ports_per_target,
    MAX(ports_scanned) as max_ports_single_target
FROM port_scans
WHERE ports_scanned > 100
GROUP BY source_ip
HAVING COUNT(DISTINCT destination_ip) > 1
ORDER BY total_ports_scanned DESC;

```

43

## 3. Anomaly Detection with Statistics

```

-- Find anomalous traffic patterns
WITH baseline AS (
    SELECT
        AVG(bytes_transferred) as avg_bytes,
        STDDEV(bytes_transferred) as stddev_bytes
    FROM network_traffic
    WHERE timestamp BETWEEN NOW() - INTERVAL '7 days' AND NOW() - INTERVAL
'1 day'

```

```

),
recent_traffic AS (
    SELECT
        source_ip,
        AVG(bytes_transferred) as recent_avg_bytes,
        COUNT(*) as connection_count
    FROM network_traffic
    WHERE timestamp > NOW() - INTERVAL '1 hour'
    GROUP BY source_ip
)
SELECT
    r.source_ip,
    r.recent_avg_bytes,
    b.avg_bytes as baseline_avg,
    (r.recent_avg_bytes - b.avg_bytes) / b.stddev_bytes as z_score,
    r.connection_count
FROM recent_traffic r
CROSS JOIN baseline b
WHERE ABS((r.recent_avg_bytes - b.avg_bytes) / b.stddev_bytes) > 3
ORDER BY ABS((r.recent_avg_bytes - b.avg_bytes) / b.stddev_bytes) DESC;

```

## Performance Tips for Aggregations

### 1. Index Strategy for GROUP BY:

44

```

CREATE INDEX idx_category_timestamp ON events(category, timestamp);
-- Helps: GROUP BY category ORDER BY timestamp

```

### 2. Materialized Views for Complex Aggregations:

```

CREATE MATERIALIZED VIEW hourly_stats AS
SELECT
    DATE_TRUNC('hour', timestamp) as hour,
    category,
    COUNT(*) as event_count,
    AVG(severity) as avg_severity
FROM security_events
GROUP BY DATE_TRUNC('hour', timestamp), category;

```

### 3. Approximate Aggregations for Speed:

```

-- PostgreSQL: approximate count
SELECT COUNT(*) FILTER (WHERE severity > 3) as high_severity_approx
FROM security_events TABLESAMPLE SYSTEM (10); -- 10% sample

```

## Common Pitfalls and Solutions

| Problem                  | Example                   | Solution                          |
|--------------------------|---------------------------|-----------------------------------|
| NULL handling            | AVG(column) ignores NULLs | Use COALESCE(column, 0) if needed |
| Division by zero         | SUM(x)/COUNT(y)           | Use NULLIF(COUNT(y), 0)           |
| String aggregation order | GROUP_CONCAT unordered    | Add ORDER BY within function      |
| Memory issues            | Large GROUP BY            | Use HAVING to filter early        |

## Memory Trick: "CS-AMM-GSH"

COUNT, SUM, AVG, MIN, MAX, GROUP\_CONCAT, STDDEV, HAVING

```

import sqlite3
import pandas as pd

# Create connection and sample database
conn = sqlite3.connect(':memory:') # In-memory database for demo
cursor = conn.cursor()

# DDL: Create tables
cursor.execute('''
    CREATE TABLE security_events (
        event_id INTEGER PRIMARY KEY AUTOINCREMENT,
        timestamp DATETIME DEFAULT CURRENT_TIMESTAMP,
        source_ip TEXT NOT NULL,
        destination_ip TEXT,
        port INTEGER,
        protocol TEXT,
        severity INTEGER CHECK (severity BETWEEN 1 AND 5),
        category TEXT,
        bytes_transferred INTEGER,
        duration_ms INTEGER
    )
''')

cursor.execute('''
    CREATE TABLE ip_reputation (
        ip_address TEXT PRIMARY KEY,
        reputation_score REAL,
        country TEXT,
        last_updated DATE
    )
''')

# Create indexes for performance
cursor.execute('CREATE INDEX idx_source_ip ON security_events(source_ip)')
cursor.execute('CREATE INDEX idx_timestamp ON security_events(timestamp)')

# DML: Insert sample data with more fields for aggregation
sample_events = [
    ('2024-01-15 10:30:00', '192.168.1.100', '10.0.0.5', 443, 'HTTPS', 2, 'web', 1024, 150),
    ('2024-01-15 10:31:00', '192.168.1.101', '10.0.0.5', 22, 'SSH', 3, 'remote', 2048, 300),
    ('2024-01-15 10:32:00', '10.0.0.99', '192.168.1.100', 3389, 'RDP', 4, 'remote', 5120, 500),
    ('2024-01-15 10:33:00', '192.168.1.100', '8.8.8.8', 53, 'DNS', 1, 'network', 512, 50),
    ('2024-01-15 10:34:00', '192.168.1.100', '10.0.0.5', 443, 'HTTPS', 2, 'web', 3072, 200),
]

```

```

        ('2024-01-15 10:35:00', '192.168.1.101', '10.0.0.5', 22, 'SSH', 3,
        'remote', 1536, 250),
    ]

cursor.executemany('''
    INSERT INTO security_events (timestamp, source_ip, destination_ip,
port, protocol,
                                severity, category, bytes_transferred,
duration_ms)
    VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
''', sample_events)

# Insert IP reputation data
ip_reputation_data = [
    ('192.168.1.100', 0.8, 'Internal', '2024-01-15'),
    ('10.0.0.99', 0.3, 'Internal', '2024-01-15'),
    ('8.8.8.8', 0.9, 'US', '2024-01-15')
]

```

```

cursor.executemany('''
    INSERT INTO ip_reputation (ip_address, reputation_score, country,
last_updated)
    VALUES (?, ?, ?, ?)
''', ip_reputation_data)

```

```
conn.commit()
```

```
# DQL: Aggregation examples
```

```

# 1. Basic aggregation functions
print("== Basic Aggregation Functions ==")
query1 = """
    SELECT
        COUNT(*) as total_events,
        COUNT(DISTINCT source_ip) as unique_sources,
        SUM(bytes_transferred) as total_bytes,
        AVG(bytes_transferred) as avg_bytes,
        MIN(bytes_transferred) as min_bytes,
        MAX(bytes_transferred) as max_bytes,
        AVG(duration_ms) as avg_duration_ms
    FROM security_events
"""

df1 = pd.read_sql_query(query1, conn)
print(df1)

```

```
# 2. Statistical aggregations
```

```

print("\n== Statistical Aggregations ==")
# SQLite doesn't have built-in STDDEV, so we calculate manually
query2 = """
    SELECT

```

```

        category,
        COUNT(*) as event_count,
        AVG(severity) as avg_severity,
        AVG(bytes_transferred) as avg_bytes,
        -- Manual variance calculation for SQLite
        AVG((bytes_transferred - (SELECT AVG(bytes_transferred) FROM
security_events)) *
            (bytes_transferred - (SELECT AVG(bytes_transferred) FROM
security_events))) as variance_bytes
    FROM security_events
    GROUP BY category
    ORDER BY avg_severity DESC
"""

df2 = pd.read_sql_query(query2, conn)
print(df2)

# 3. Advanced aggregation with GROUP_CONCAT (SQLite version)
print("\n== Advanced Aggregation ==")
query3 = """
    SELECT
        source_ip,
        COUNT(*) as connection_count,
        GROUP_CONCAT(DISTINCT protocol) as protocols_used,
        GROUP_CONCAT(DISTINCT destination_ip) as destinations,
        SUM(bytes_transferred) as total_bytes_sent,
        AVG(severity) as avg_threat_level
    FROM security_events
    GROUP BY source_ip
    HAVING COUNT(*) > 1
    ORDER BY total_bytes_sent DESC
"""

df3 = pd.read_sql_query(query3, conn)
print(df3)

# 4. Window functions example
print("\n== Window Functions ==")
query4 = """
    SELECT
        timestamp,
        source_ip,
        bytes_transferred,
        SUM(bytes_transferred) OVER (PARTITION BY source_ip ORDER BY
timestamp) as cumulative_bytes,
        ROW_NUMBER() OVER (PARTITION BY source_ip ORDER BY timestamp) as
event_sequence,
        RANK() OVER (ORDER BY bytes_transferred DESC) as size_rank,
        LAG(bytes_transferred, 1) OVER (PARTITION BY source_ip ORDER BY
timestamp) as prev_bytes
    FROM security_events
    ORDER BY source_ip, timestamp
"""

```

```
"""
df4 = pd.read_sql_query(query4, conn)
print(df4)

# 5. Percentile calculation (using window functions)
print("\n==== Percentile Analysis ===")
query5 = """
    WITH ranked_events AS (
        SELECT
            category,
            bytes_transferred,
            PERCENT_RANK() OVER (PARTITION BY category ORDER BY
bytes_transferred) as percentile
        FROM security_events
    )
    SELECT
        category,
        MAX(CASE WHEN percentile <= 0.25 THEN bytes_transferred END) as
p25,
        MAX(CASE WHEN percentile <= 0.50 THEN bytes_transferred END) as
p50_median,
        MAX(CASE WHEN percentile <= 0.75 THEN bytes_transferred END) as
p75,
        MAX(CASE WHEN percentile <= 0.95 THEN bytes_transferred END) as
p95
        FROM ranked_events
        GROUP BY category
"""

df5 = pd.read_sql_query(query5, conn)
print(df5)

# Close connection
conn.close()
```

# Introduction to Matplotlib

Matplotlib is the foundational plotting library in Python, providing:

- **Complete control** over every aspect of a figure
- **Publication-quality** plots
- **Extensive customization** options
- **Multiple backends** for different output formats

## Core Plotting Functions

### Essential Plot Types

| Function               | Purpose                 | Use Case                    |
|------------------------|-------------------------|-----------------------------|
| plot()                 | Line plots              | Time series, trends         |
| scatter()              | Scatter plots           | Relationships, correlations |
| bar() / barh()         | Bar charts              | Categorical comparisons     |
| hist()                 | Histograms              | Distribution analysis       |
| pie()                  | Pie charts              | Proportions (use sparingly) |
| imshow()               | Display images/matrices | Heatmaps, image data        |
| contour() / contourf() | Contour plots           | 3D data on 2D plane         |
| boxplot()              | Box plots               | Statistical summaries       |

50

## Figure and Axes Management

| Function   | Purpose                    | Example                                     |
|------------|----------------------------|---|
| figure()   | Create new figure          | <code>plt.figure(figsize=(10, 6))</code>    |
| subplot()  | Create subplot             | <code>plt.subplot(2, 2, 1)</code>           |
| subplots() | Create figure and subplots | <code>fig, axes = plt.subplots(2, 3)</code> |
| axes()     | Add axes at position       | <code>plt.axes([0.1, 0.1, 0.8, 0.8])</code> |

## Styling and Formatting

| Function            | Purpose         | Example                                   |
|---------------------|-----------------|---|
| xlabel() / ylabel() | Set axis labels | <code>plt.xlabel('Time (s)')</code>       |
| title()             | Set plot title  | <code>plt.title('Network Traffic')</code> |

| Function            | Purpose              | Example                       |
|---------------------|----------------------|-------------------------------|
| legend()            | Add legend           | plt.legend(['Train', 'Test']) |
| xlim() / ylim()     | Set axis limits      | plt.xlim(0, 100)              |
| xticks() / yticks() | Customize tick marks | plt.xticks(rotation=45)       |
| grid()              | Add gridlines        | plt.grid(True, alpha=0.3)     |
| text() / annotate() | Add text/annotations | plt.text(x, y, 'Peak')        |

## Display and Save

| Function       | Purpose             | Example                          |
|----------------|---------------------|----------------------------------|
| show()         | Display all figures | plt.show()                       |
| savefig()      | Save to file        | plt.savefig('plot.png', dpi=300) |
| close()        | Close figure        | plt.close('all')                 |
| tight_layout() | Auto-adjust spacing | plt.tight_layout()               |
| colorbar()     | Add colorbar        | plt.colorbar()                   |
| clf() / cla()  | Clear figure/axes   | plt.clf()                        |

## Matplotlib for Machine Learning

51

```

import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
from matplotlib import cm

# Set style for better-looking plots
plt.style.use('seaborn-v0_8') # or 'ggplot', 'default'
plt.rcParams['figure.figsize'] = (10, 6)
plt.rcParams['font.size'] = 12

# 1. Data Exploration Examples
def explore_security_data():
    """Demonstrate matplotlib for security data exploration"""

    # Generate sample security data
    np.random.seed(42)
    n_events = 1000

    # Event types and their counts
    event_types = ['Login Attempt', 'Port Scan', 'File Access', 'Network
Traffic', 'System Call']
    event_counts = np.random.randint(50, 300, size=len(event_types))

```

```

# Time series data
hours = np.arange(24)
traffic_volume = 100 + 50 * np.sin(hours * np.pi / 12) +
np.random.normal(0, 10, 24)
attack_events = np.random.poisson(5, 24)

# Create figure with subplots
fig, axes = plt.subplots(2, 2, figsize=(15, 10))

# 1. Bar chart for event types
ax1 = axes[0, 0]
bars = ax1.bar(event_types, event_counts, color='skyblue',
edgecolor='navy', alpha=0.7)
ax1.set_xlabel('Event Type')
ax1.set_ylabel('Count')
ax1.set_title('Security Event Distribution')
ax1.tick_params(axis='x', rotation=45)

# Add value labels on bars
for bar in bars:
    height = bar.get_height()
    ax1.text(bar.get_x() + bar.get_width()/2., height,
             f'{int(height)}', ha='center', va='bottom')

# 2. Time series plot
ax2 = axes[0, 1]
ax2.plot(hours, traffic_volume, 'b-', linewidth=2, label='Normal
Traffic')
ax2.bar(hours, attack_events * 10, color='red', alpha=0.5,
label='Attack Events')
ax2.set_xlabel('Hour of Day')
ax2.set_ylabel('Volume')
ax2.set_title('24-Hour Traffic Pattern')
ax2.legend()
ax2.grid(True, alpha=0.3)

# 3. Scatter plot with correlation
ports = np.random.randint(1, 65535, n_events)
severity = 5 * np.exp(-ports/10000) + np.random.normal(0, 0.5,
n_events)
severity = np.clip(severity, 0, 5)

ax3 = axes[1, 0]
scatter = ax3.scatter(ports, severity, alpha=0.5, c=severity,
cmap='RdYlBu_r', s=20)
ax3.set_xlabel('Port Number')
ax3.set_ylabel('Severity Score')
ax3.set_title('Port vs Threat Severity')
ax3.set_xlim(0, 65535)

```

```

plt.colorbar(scatter, ax=ax3, label='Severity')

# 4. Histogram for distribution analysis
ax4 = axes[1, 1]
response_times = np.random.gamma(2, 2, 1000)
n, bins, patches = ax4.hist(response_times, bins=30, alpha=0.7,
color='green', edgecolor='black')

# Highlight outliers
outlier_threshold = np.percentile(response_times, 95)
for i, patch in enumerate(patches):
    if bins[i] >= outlier_threshold:
        patch.set_facecolor('red')

ax4.axvline(np.mean(response_times), color='red', linestyle='--',
linewidth=2, label=f'Mean: {np.mean(response_times):.2f}')
ax4.axvline(np.median(response_times), color='blue', linestyle='--',
linewidth=2, label=f'Median: {np.median(response_times):.2f}')
ax4.set_xlabel('Response Time (ms)')
ax4.set_ylabel('Frequency')
ax4.set_title('Response Time Distribution')
ax4.legend()

plt.tight_layout()
plt.show()

```

53

```

# 2. Model Performance Visualization
def visualize_model_performance():
    """Demonstrate model evaluation plots"""

    fig, axes = plt.subplots(2, 3, figsize=(18, 12))

    # 1. Confusion Matrix
    ax1 = axes[0, 0]
    cm = np.array([[85, 15, 0], [20, 75, 5], [1, 4, 95]])
    im = ax1.imshow(cm, cmap='Blues')

    # Add colorbar
    plt.colorbar(im, ax=ax1)

    # Add labels
    classes = ['Normal', 'Warning', 'Attack']
    ax1.set_xticks(np.arange(len(classes)))
    ax1.set_yticks(np.arange(len(classes)))
    ax1.set_xticklabels(classes)
    ax1.set_yticklabels(classes)
    ax1.set_xlabel('Predicted')
    ax1.set_ylabel('Actual')
    ax1.set_title('Confusion Matrix')

```

```

# Add text annotations
for i in range(len(classes)):
    for j in range(len(classes)):
        text = ax1.text(j, i, cm[i, j], ha="center", va="center",
                        color="white" if cm[i, j] > cm.max()/2 else
"black")

# 2. ROC Curve
ax2 = axes[0, 1]
fpr = np.array([0., 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.])
tpr = np.array([0., 0.35, 0.5, 0.65, 0.75, 0.85, 0.9, 0.93, 0.95,
0.97, 1.])

ax2.plot(fpr, tpr, 'b-', linewidth=2, label='Model (AUC = 0.85)')
ax2.plot([0, 1], [0, 1], 'r--', linewidth=2, label='Random (AUC =
0.50)')
ax2.fill_between(fpr, tpr, alpha=0.3)
ax2.set_xlabel('False Positive Rate')
ax2.set_ylabel('True Positive Rate')
ax2.set_title('ROC Curve')
ax2.legend()
ax2.grid(True, alpha=0.3)

# 3. Learning Curves
ax3 = axes[0, 2]
epochs = np.arange(1, 101)
train_loss = 2.5 * np.exp(-epochs/30) + 0.1 + np.random.normal(0,
0.02, 100)
val_loss = 2.5 * np.exp(-epochs/25) + 0.15 + np.random.normal(0, 0.03,
100)

ax3.plot(epochs, train_loss, 'b-', label='Training Loss')
ax3.plot(epochs, val_loss, 'r-', label='Validation Loss')
ax3.set_xlabel('Epoch')
ax3.set_ylabel('Loss')
ax3.set_title('Training History')
ax3.legend()
ax3.grid(True, alpha=0.3)

# 4. Feature Importance
ax4 = axes[1, 0]
features = ['Port Number', 'Packet Size', 'Duration', 'Protocol',
'Flags', 'Bytes Sent', 'Bytes Received']
importance = np.array([0.25, 0.18, 0.15, 0.12, 0.10, 0.12, 0.08])

y_pos = np.arange(len(features))
ax4.barh(y_pos, importance, color='green', alpha=0.7)
ax4.set_yticks(y_pos)
ax4.set_yticklabels(features)
ax4.set_xlabel('Importance Score')

```

```

ax4.set_title('Feature Importance')
ax4.grid(True, alpha=0.3, axis='x')

# 5. Precision-Recall Curve
ax5 = axes[1, 1]
recall = np.array([1., 0.95, 0.9, 0.85, 0.8, 0.7, 0.6, 0.5, 0.3, 0.1,
0.])
precision = np.array([0.5, 0.55, 0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9,
0.95, 1.])

ax5.plot(recall, precision, 'g-', linewidth=2)
ax5.fill_between(recall, precision, alpha=0.3, color='green')
ax5.set_xlabel('Recall')
ax5.set_ylabel('Precision')
ax5.set_title('Precision-Recall Curve')
ax5.grid(True, alpha=0.3)

# 6. Residual Plot
ax6 = axes[1, 2]
y_true = np.random.normal(100, 20, 200)
y_pred = y_true + np.random.normal(0, 10, 200)
residuals = y_true - y_pred

ax6.scatter(y_pred, residuals, alpha=0.5)
ax6.axhline(y=0, color='red', linestyle='--')
ax6.set_xlabel('Predicted Values')
ax6.set_ylabel('Residuals')
ax6.set_title('Residual Plot')
ax6.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

```

```

# 3. Advanced Visualization Techniques
def advanced_visualizations():
    """Demonstrate advanced matplotlib features"""

    # Create complex layout
    fig = plt.figure(figsize=(16, 10))

    # Custom grid layout
    gs = fig.add_gridspec(3, 3, hspace=0.3, wspace=0.3)

    # 1. 3D Surface Plot (using projection)
    ax1 = fig.add_subplot(gs[0:2, 0:2], projection='3d')
    X = np.arange(-5, 5, 0.25)
    Y = np.arange(-5, 5, 0.25)
    X, Y = np.meshgrid(X, Y)
    R = np.sqrt(X**2 + Y**2)
    Z = np.sin(R)

```

```

surf = ax1.plot_surface(X, Y, Z, cmap='viridis', alpha=0.8)
ax1.set_xlabel('X')
ax1.set_ylabel('Y')
ax1.set_zlabel('Z')
ax1.set_title('3D Surface Plot')

# 2. Heatmap with annotations
ax2 = fig.add_subplot(gs[0, 2])
data = np.random.randn(5, 5)
im = ax2.imshow(data, cmap='coolwarm', aspect='auto')

# Add grid
ax2.set_xticks(np.arange(5))
ax2.set_yticks(np.arange(5))
ax2.set_xticklabels(['A', 'B', 'C', 'D', 'E'])
ax2.set_yticklabels(['1', '2', '3', '4', '5'])

# Add text annotations
for i in range(5):
    for j in range(5):
        text = ax2.text(j, i, f'{data[i, j]:.2f}', ha="center", va="center", color="black",
fontsize=10)

ax2.set_title('Correlation Heatmap')
plt.colorbar(im, ax=ax2)

# 3. Multiple Y-axes
ax3 = fig.add_subplot(gs[1, 2])
x = np.arange(0, 10, 0.1)
y1 = np.sin(x)
y2 = 100 * np.cos(x)

color = 'tab:blue'
ax3.set_xlabel('Time')
ax3.set_ylabel('Amplitude', color=color)
ax3.plot(x, y1, color=color)
ax3.tick_params(axis='y', labelcolor=color)

ax3_twin = ax3.twinx()
color = 'tab:red'
ax3_twin.set_ylabel('Power', color=color)
ax3_twin.plot(x, y2, color=color)
ax3_twin.tick_params(axis='y', labelcolor=color)

ax3.set_title('Dual Y-Axis Plot')
ax3.grid(True, alpha=0.3)

# 4. Subplots with shared axes

```

```

ax4 = fig.add_subplot(gs[2, :])

# Generate time series data
dates = pd.date_range('2024-01-01', periods=100, freq='D')
values1 = np.cumsum(np.random.randn(100)) + 100
values2 = np.cumsum(np.random.randn(100)) + 100
values3 = np.cumsum(np.random.randn(100)) + 100

ax4.plot(dates, values1, label='Metric 1')
ax4.plot(dates, values2, label='Metric 2')
ax4.plot(dates, values3, label='Metric 3')
ax4.set_xlabel('Date')
ax4.set_ylabel('Value')
ax4.set_title('Time Series Comparison')
ax4.legend()
ax4.grid(True, alpha=0.3)

# Format x-axis dates
fig.autofmt_xdate()

plt.tight_layout()
plt.show()

# Run the visualization examples
if __name__ == "__main__":
    print("Exploring Security Data...")
    explore_security_data()

    print("\nVisualizing Model Performance...")
    visualize_model_performance()

    print("\nAdvanced Visualizations...")
    advanced_visualizations()

```

57

## Matplotlib Best Practices

1. **Always label your axes** - Units matter in ML
2. **Use appropriate plot types** - Bar for categories, line for trends
3. **Add legends when multiple series** - Make plots self-explanatory
4. **Consider colorblind-friendly palettes** - Use 'viridis' or 'cividis'
5. **Save high-resolution for publications** - dpi=300 or higher
6. **Use consistent styling** - Set rcParams at start

## Common Pitfalls to Avoid

- **Overplotting:** Use alpha transparency or sampling for large datasets
- **3D plots for 2D data:** Often misleading, use 2D with color instead

- **Too many colors:** Limit to 5-7 distinct colors maximum
  - **Missing error bars:** Always show uncertainty when relevant
  - **Chartjunk:** Remove unnecessary decorations
-

# Understanding NoSQL

## NoSQL vs SQL: When to Use Each

| Aspect         | SQL (Relational)         | NoSQL (MongoDB)                |
|----------------|--------------------------|--------------------------------|
| Data Structure | Fixed schema, tables     | Flexible schema, documents     |
| Relationships  | Foreign keys, JOINS      | Embedded documents, references |
| Scaling        | Vertical (bigger server) | Horizontal (more servers)      |
| ACID           | Full ACID compliance     | Eventual consistency*          |
| Query Language | SQL standard             | Database-specific              |
| Use Cases      | Transactions, reporting  | Big data, real-time, flexible  |

\*Modern MongoDB supports ACID transactions

## MongoDB Concepts

| SQL Term | → | MongoDB Equivalent            |
|----------|---|-------------------------------|
| Database | → | Database                      |
| Table    | → | Collection                    |
| Row      | → | Document                      |
| Column   | → | Field                         |
| Index    | → | Index                         |
| JOIN     | → | Embedded documents / \$lookup |

59

## MongoDB Query Operations

### CRUD Operations

| Operation | SQL         | MongoDB                   |
|-----------|-------------|---------------------------|
| Create    | INSERT INTO | db.collection.insertOne() |
| Read      | SELECT      | db.collection.find()      |
| Update    | UPDATE      | db.collection.updateOne() |
| Delete    | DELETE      | db.collection.deleteOne() |

### Query Operators

| Operator     | Purpose                 | Example                            |
|--------------|-------------------------|------------------------------------|
| \$eq , \$ne  | Equal, not equal        | {age: {\$eq: 25}}                  |
| \$gt , \$gte | Greater than (or equal) | {score: {\$gt: 80}}                |
| \$lt , \$lte | Less than (or equal)    | {price: {\$lte: 100}}              |
| \$in , \$nin | In array, not in array  | {status: {\$in: ['A', 'B']}}       |
| \$and , \$or | Logical operators       | {\$or: [{age: 25}, {status: 'A'}]} |
| \$regex      | Pattern matching        | {name: {\$regex: '^John'}}         |
| \$exists     | Field exists            | {email: {\$exists: true}}          |

## MongoDB Aggregation Pipeline

The aggregation pipeline is MongoDB's powerful data processing framework:

```
pipeline = [
    {"$match": {...}},      # Filter documents
    {"$group": {...}},      # Group and aggregate
    {"$sort": {...}},       # Sort results
    {"$project": {...}},    # Shape output
    {"$limit": n}           # Limit results
]
```

60

## Practical MongoDB Implementation

```
from pymongo import MongoClient
from datetime import datetime, timedelta
import pandas as pd
from bson import ObjectId

# Connect to MongoDB
client = MongoClient('mongodb://localhost:27017/')
db = client['security_analytics']

# Create collections
events_collection = db['security_events']
threats_collection = db['threat_intelligence']

# Create indexes for performance
events_collection.create_index([('timestamp', -1)])
events_collection.create_index([('source_ip', 1)])
events_collection.create_index([('severity', -1), ('timestamp', -1)])

# 1. Insert Documents
def insert_security_events():
    pass
```

```

"""Insert sample security events"""

events = [
    {
        "_id": ObjectId(),
        "timestamp": datetime.now(),
        "event_type": "failed_login",
        "source_ip": "192.168.1.100",
        "target_ip": "10.0.0.5",
        "user": "admin",
        "metadata": {
            "attempts": 5,
            "location": "US",
            "user_agent": "Mozilla/5.0"
        },
        "severity": 3,
        "tags": ["brute_force", "suspicious"]
    },
    {
        "_id": ObjectId(),
        "timestamp": datetime.now() - timedelta(hours=1),
        "event_type": "port_scan",
        "source_ip": "10.0.0.99",
        "target_ip": "192.168.1.0/24",
        "ports_scanned": [22, 80, 443, 3306, 3389],
        "metadata": {
            "scan_type": "SYN",
            "duration_seconds": 120
        },
        "severity": 4,
        "tags": ["reconnaissance", "automated"]
    },
    {
        "_id": ObjectId(),
        "timestamp": datetime.now() - timedelta(hours=2),
        "event_type": "malware_detected",
        "source_ip": "192.168.1.150",
        "file_hash": "d41d8cd98f00b204e9800998ecf8427e",
        "malware_family": "Emotet",
        "metadata": {
            "file_name": "invoice.pdf.exe",
            "detection_engine": "ClamAV",
            "confidence": 0.95
        },
        "severity": 5,
        "tags": ["malware", "trojan", "critical"]
    }
]

# Insert multiple documents

```

```

result = events_collection.insert_many(events)
print(f"Inserted {len(result.inserted_ids)} events")

# Insert threat intelligence
threats = [
    {
        "ip_address": "10.0.0.99",
        "reputation_score": 0.2,
        "threat_type": "scanner",
        "first_seen": datetime.now() - timedelta(days=30),
        "last_seen": datetime.now(),
        "associated_malware": [],
        "tags": ["automated", "persistent"]
    }
]

threats_collection.insert_many(threats)

# 2. Query Documents
def query_examples():
    """Demonstrate various MongoDB queries"""

    print("==== Basic Queries ===")

    # Find all high severity events
    high_severity = events_collection.find({"severity": {"$gte": 4}})
    print(f"High severity events: {high_severity.count()}") 62

    # Find events by IP with projection
    ip_events = events_collection.find(
        {"source_ip": "192.168.1.100"},
        {"timestamp": 1, "event_type": 1, "severity": 1}
    )

    for event in ip_events:
        print(f"Event: {event}")

    # Complex query with multiple conditions
    complex_query = events_collection.find({
        "$and": [
            {"severity": {"$gte": 3}},
            {"timestamp": {"$gte": datetime.now() - timedelta(hours=24)}},
            {"tags": {"$in": ["suspicious", "critical"]}}
        ]
    }).sort("timestamp", -1)

    print("\n==== Recent Suspicious Events ===")
    for event in complex_query.limit(5):
        print(f"{event['timestamp']}: {event['event_type']} - Severity: {event['severity']}")

```

```

# 3. Aggregation Pipeline
def aggregation_examples():
    """Demonstrate MongoDB aggregation framework"""

    print("\n==== Aggregation Examples ===")

    # Pipeline 1: Group by event type and calculate statistics
    pipeline1 = [
        {
            "$group": {
                "_id": "$event_type",
                "count": {"$sum": 1},
                "avg_severity": {"$avg": "$severity"},
                "max_severity": {"$max": "$severity"},
                "unique_ips": {"$addToSet": "$source_ip"}
            }
        },
        {
            "$project": {
                "event_type": "$_id",
                "count": 1,
                "avg_severity": {"$round": ["$avg_severity", 2]},
                "max_severity": 1,
                "unique_ip_count": {"$size": "$unique_ips"}
            }
        },
        {
            "$sort": {"avg_severity": -1}
        }
    ]

    results = events_collection.aggregate(pipeline1)
    print("\nEvent Type Statistics:")
    for result in results:
        print(result)

    # Pipeline 2: Time-based aggregation
    pipeline2 = [
        {
            "$match": {
                "timestamp": {"$gte": datetime.now() - timedelta(days=7)}
            }
        },
        {
            "$group": {
                "_id": {
                    "$dateToString": {
                        "format": "%Y-%m-%d %H:00",
                        "date": "$timestamp"
                    }
                }
            }
        }
    ]

```

```
        }
    },
    "event_count": {"$sum": 1},
    "severity_sum": {"$sum": "$severity"},
    "event_types": {"$push": "$event_type"}
}
},
{
    "$sort": {"_id": 1}
}
]

hourly_stats = events_collection.aggregate(pipeline2)
print("\nHourly Statistics:")
for stat in hourly_stats:
    print(f"{stat['_id']}: {stat['event_count']} events")

# Pipeline 3: Join with threat intelligence
pipeline3 = [
    {
        "$lookup": {
            "from": "threat_intelligence",
            "localField": "source_ip",
            "foreignField": "ip_address",
            "as": "threat_info"
        }
    },
    {
        "$unwind": {
            "path": "$threat_info",
            "preserveNullAndEmptyArrays": True
        }
    },
    {
        "$project": {
            "timestamp": 1,
            "event_type": 1,
            "source_ip": 1,
            "severity": 1,
            "threat_score": {
                "$ifNull": ["$threat_info.reputation_score", 1.0]
            }
        }
    },
    {
        "$match": {
            "threat_score": {"$lt": 0.5}
        }
    }
]
```

```

threat_events = events_collection.aggregate(pipeline3)
print("\nEvents from Known Threats:")
for event in threat_events:
    print(f"IP: {event['source_ip']} - Score: {event['threat_score']}")

# 4. Updates and Deletes
def update_examples():
    """Demonstrate update operations"""

    # Update single document
    result = events_collection.update_one(
        {"source_ip": "192.168.1.100"},
        {
            "$set": {"reviewed": True, "reviewer": "analyst1"},
            "$inc": {"severity": 1},
            "$push": {"tags": "reviewed"}
        }
    )
    print(f"Modified {result.modified_count} document(s)")

    # Update multiple documents
    result = events_collection.update_many(
        {"severity": {"$gte": 4}},
        {"$set": {"priority": "high"}}
    )
    print(f"Modified {result.modified_count} document(s)")

# 5. Text Search and Geospatial
def advanced_features():
    """Demonstrate advanced MongoDB features"""

    # Create text index
    events_collection.create_index([("event_type", "text"), ("tags", "text")])

    # Text search
    text_results = events_collection.find(
        {"$text": {"$search": "malware suspicious"}}
    )

    print("Text Search Results:")
    for result in text_results:
        print(f"Found: {result['event_type']}")

    # Create geospatial collection for IP locations
    geo_collection = db['ip_locations']
    geo_collection.create_index([("location", "2dsphere")])

```

```

# Insert location data
geo_collection.insert_one({
    "ip": "192.168.1.100",
    "location": {
        "type": "Point",
        "coordinates": [-73.97, 40.77] # longitude, latitude
    },
    "city": "New York"
})

# Find nearby IPs
nearby = geo_collection.find({
    "location": {
        "$near": {
            "$geometry": {"type": "Point", "coordinates": [-73.97,
40.77]},
            "$maxDistance": 1000 # meters
        }
    }
})

# Convert MongoDB cursor to DataFrame
def mongodb_to_dataframe():
    """Convert MongoDB query results to pandas DataFrame"""

    # Query all events
    cursor = events_collection.find()

    # Convert to DataFrame
    df = pd.DataFrame(list(cursor))

    # Process nested fields
    if 'metadata' in df.columns:
        # Normalize nested metadata
        metadata_df = pd.json_normalize(df['metadata'])
        df = pd.concat([df.drop('metadata', axis=1), metadata_df], axis=1)

    print("\nDataFrame from MongoDB:")
    print(df.head())

    return df

# Run examples
if __name__ == "__main__":
    # Clear existing data
    events_collection.delete_many({})
    threats_collection.delete_many({})

    # Run examples
    insert_security_events()

```

```
query_examples()
aggregation_examples()
update_examples()
advanced_features()
df = mongodb_to_dataframe()

# Close connection
client.close()
```

---

# Ethical Considerations

Before scraping any website, always:

1. **Check robots.txt:** Respect the site's scraping policies
2. **Read Terms of Service:** Ensure scraping is allowed
3. **Rate limit requests:** Don't overwhelm servers
4. **Identify yourself:** Use proper User-Agent headers
5. **Cache responses:** Avoid redundant requests

## Web Scraping Tools Comparison

| Tool                     | Best For        | Speed     | JavaScript | Learning Curve |
|--------------------------|-----------------|-----------|------------|----------------|
| requests + BeautifulSoup | Static HTML     | Fast      | No         | Easy           |
| Scrapy                   | Large projects  | Very Fast | No         | Moderate       |
| Selenium                 | Dynamic content | Slow      | Yes        | Moderate       |
| Playwright               | Modern web apps | Medium    | Yes        | Moderate       |
| requests-html            | Mixed content   | Medium    | Yes*       | Easy           |

68

\*Limited JavaScript support

## HTTP Headers for Responsible Scraping

```
headers = {
    'User-Agent': 'Mozilla/5.0 (Your Bot Name)',
    'Accept': 'text/html,application/xhtml+xml',
    'Accept-Language': 'en-US,en;q=0.9',
    'Accept-Encoding': 'gzip, deflate',
    'DNT': '1',
    'Connection': 'keep-alive',
    'Referer': 'https://google.com'
}
```

## Practical Web Scraping Implementation

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

```
import json
from urllib.parse import urljoin, urlparse
import pandas as pd
from requests.adapters import HTTPAdapter
from requests.packages.urllib3.util.retry import Retry

class EthicalScraper:
    """A responsible web scraper with rate limiting and error handling"""

    def __init__(self, base_url, delay=1.0):
        self.base_url = base_url
        self.delay = delay
        self.session = self._create_session()
        self.robots_rules = self._check_robots()

    def _create_session(self):
        """Create a session with retry logic"""
        session = requests.Session()

        # Retry strategy
        retry = Retry(
            total=3,
            read=3,
            connect=3,
            backoff_factor=0.3,
            status_forcelist=(500, 502, 504)
        )

        adapter = HTTPAdapter(max_retries=retry)
        session.mount('http://', adapter)
        session.mount('https://', adapter)

        # Set headers
        session.headers.update({
            'User-Agent': 'EthicalScraper/1.0 (Educational Purpose)'
        })

        return session

    def _check_robots(self):
        """Check robots.txt for scraping rules"""
        robots_url = urljoin(self.base_url, '/robots.txt')
        try:
            response = self.session.get(robots_url)
            if response.status_code == 200:
                print(f"Robots.txt found at {robots_url}")
                return response.text
        except:
            print("No robots.txt found")
        return None
```

```
def scrape_page(self, url):
    """Scrape a single page with rate limiting"""
    time.sleep(self.delay) # Rate limiting

    try:
        response = self.session.get(url, timeout=10)
        response.raise_for_status()
        return response
    except requests.exceptions.RequestException as e:
        print(f"Error scraping {url}: {e}")
        return None

def parse_security_advisories(self, html):
    """Parse security advisory data from HTML"""
    soup = BeautifulSoup(html, 'html.parser')
    advisories = []

    # Example: Parse CVE information
    for article in soup.find_all('article', class_='advisory'):
        advisory = {}

        # Extract CVE ID
        cve_element = article.find('span', class_='cve-id')
        if cve_element:
            advisory['cve_id'] = cve_element.text.strip() 70

        # Extract severity
        severity_element = article.find('span', class_='severity')
        if severity_element:
            advisory['severity'] = severity_element.text.strip()

        # Extract description
        desc_element = article.find('div', class_='description')
        if desc_element:
            advisory['description'] = desc_element.text.strip()

        # Extract affected products
        products = article.find_all('li', class_='product')
        advisory['affected_products'] = [p.text.strip() for p in
products]

        # Extract dates
        date_element = article.find('time')
        if date_element:
            advisory['published_date'] = date_element.get('datetime')

        advisories.append(advisory)

    return advisories
```

```

# Example: Scraping threat intelligence feeds
def scrape_threat_feeds():
    """Example of scraping multiple threat intelligence sources"""

    # Example threat feed sources (use actual URLs)
    feeds = [
        {
            'name': 'Example Threat Feed',
            'url': 'https://example.com/threats',
            'parser': 'json'
        },
        {
            'name': 'CVE Database',
            'url': 'https://example.com/cve',
            'parser': 'html'
        }
    ]

    all_threats = []

    for feed in feeds:
        print(f"Scraping {feed['name']}...")
        scraper = EthicalScraper(feed['url'])

        response = scraper.scrape_page(feed['url'])
        if response:
            if feed['parser'] == 'json':
                # Parse JSON response
                data = response.json()
                threats = data.get('threats', [])
                all_threats.extend(threats)

            elif feed['parser'] == 'html':
                # Parse HTML response
                threats = scraper.parse_security_advisories(response.text)
                all_threats.extend(threats)

    # Convert to DataFrame
    df = pd.DataFrame(all_threats)
    return df

# Advanced scraping with Selenium for dynamic content
def scrape_dynamic_content():
    """Scrape JavaScript-rendered content using Selenium"""
    from selenium import webdriver
    from selenium.webdriver.common.by import By
    from selenium.webdriver.support.ui import WebDriverWait
    from selenium.webdriver.support import expected_conditions as EC
    from selenium.webdriver.chrome.options import Options

```

```
# Configure Chrome options
chrome_options = Options()
chrome_options.add_argument('--headless') # Run in background
chrome_options.add_argument('--no-sandbox')
chrome_options.add_argument('--disable-dev-shm-usage')

# Initialize driver
driver = webdriver.Chrome(options=chrome_options)

try:
    # Navigate to page
    driver.get('https://example.com/dynamic-content')

    # Wait for dynamic content to load
    wait = WebDriverWait(driver, 10)
    wait.until(EC.presence_of_element_located((By.CLASS_NAME, 'threat-data')))

    # Extract data after JavaScript execution
    threats = []
    threat_elements = driver.find_elements(By.CLASS_NAME, 'threat-item')

    for element in threat_elements:
        threat = {
            'title': element.find_element(By.CLASS_NAME, 'title').text,
            'severity': element.find_element(By.CLASS_NAME, 'severity').text,
            'date': element.find_element(By.CLASS_NAME, 'date').text
        }
        threats.append(threat)

    return threats

finally:
    driver.quit()

# Scraping with respect to rate limits
class RateLimitedScraper:
    """Scraping with advanced rate limiting"""

    def __init__(self, requests_per_second=1):
        self.min_interval = 1.0 / requests_per_second
        self.last_request_time = 0

    def throttle(self):
        """Implement rate limiting"""
        current_time = time.time()
```

```

        time_since_last = current_time - self.last_request_time

        if time_since_last < self.min_interval:
            sleep_time = self.min_interval - time_since_last
            time.sleep(sleep_time)

    self.last_request_time = time.time()

def scrape_with_pagination(self, base_url, max_pages=10):
    """Scrape paginated content"""
    all_data = []

    for page in range(1, max_pages + 1):
        self.throttle() # Rate limit

        url = f"{base_url}?page={page}"
        response = requests.get(url)

        if response.status_code == 200:
            soup = BeautifulSoup(response.text, 'html.parser')

            # Extract data from page
            items = soup.find_all('div', class_='item')
            for item in items:
                data = self.extract_item_data(item)
                all_data.append(data)

            # Check if there's a next page
            next_link = soup.find('a', {'rel': 'next'})
            if not next_link:
                break
            else:
                print(f"Failed to scrape page {page}")
                break

    return all_data

def extract_item_data(self, item):
    """Extract data from a single item"""
    return {
        'title': item.find('h3').text.strip() if item.find('h3') else '',
        'description': item.find('p').text.strip() if item.find('p') else '',
        'link': item.find('a')['href'] if item.find('a') else ''
    }

# Example usage
if __name__ == "__main__":
    # Basic scraping

```

```
scraper = EthicalScraper('https://example.com', delay=2.0)
response = scraper.scrape_page('https://example.com/security-advisories')

if response:
    advisories = scraper.parse_security_advisories(response.text)
    df = pd.DataFrame(advisories)
    print(f"Scraped {len(df)} advisories")
    print(df.head())

# Rate-limited scraping
rate_scraper = RateLimitedScraper(requests_per_second=0.5)
paginated_data =
rate_scraper.scrape_with_pagination('https://example.com/threats')
print(f"Scraped {len(paginated_data)} items across multiple pages")
```

---

# Common Data Quality Issues

| Issue                      | Description        | Detection Method    | Solution               |
|----------------------------|--------------------|---------------------|------------------------|
| <b>Missing Values</b>      | NaN, NULL, empty   | .isnull() , .isna() | Imputation, deletion   |
| <b>Duplicates</b>          | Repeated records   | .duplicated()       | Remove or merge        |
| <b>Outliers</b>            | Extreme values     | IQR, Z-score        | Cap, transform, remove |
| <b>Inconsistent Format</b> | Mixed date formats | Regex patterns      | Standardize            |
| <b>Invalid Data</b>        | Out of range       | Validation rules    | Correct or remove      |
| <b>Encoding Issues</b>     | Character errors   | Try/except decode   | Fix encoding           |

## Missing Data Strategies

```
# Missing data patterns
Missing Completely at Random (MCAR)
└─ No pattern to missingness
└─ Safe to delete
└─ Example: Random sensor failures

Missing at Random (MAR)
└─ Missingness depends on other variables
└─ Imputation preferred
└─ Example: Age missing for certain demographics

Missing Not at Random (MNAR)
└─ Missingness depends on value itself
└─ Requires domain knowledge
└─ Example: High values not recorded due to sensor limits
```

75

## Comprehensive Data Cleaning Implementation

```
import pandas as pd
import numpy as np
from scipy import stats
import re
from datetime import datetime

class DataCleaner:
    """Comprehensive data cleaning utilities for security data"""

    def __init__(self, df):
```

```
self.df = df.copy()
self.cleaning_report = {}

def generate_report(self):
    """Generate data quality report"""
    report = {
        'shape': self.df.shape,
        'missing_values': self.df.isnull().sum().to_dict(),
        'duplicates': self.df.duplicated().sum(),
        'data_types': self.df.dtypes.to_dict(),
        'numeric_summary': self.df.describe().to_dict()
    }

    # Check for potential issues
    issues = []

    # High missing percentage
    for col, missing in report['missing_values'].items():
        missing_pct = (missing / len(self.df)) * 100
        if missing_pct > 20:
            issues.append(f"Column '{col}' has {missing_pct:.1f}% missing values")

    # Check for high cardinality in categorical columns
    for col in self.df.select_dtypes(include=['object']).columns:
        unique_ratio = self.df[col].nunique() / len(self.df)
        if unique_ratio > 0.5:
            issues.append(f"Column '{col}' has high cardinality ({unique_ratio:.2%})")

    report['issues'] = issues
    return report

def handle_missing_values(self, strategy='auto'):
    """Handle missing values with various strategies"""

    for col in self.df.columns:
        missing_count = self.df[col].isnull().sum()

        if missing_count > 0:
            missing_pct = (missing_count / len(self.df)) * 100

            if strategy == 'auto':
                # Automatic strategy selection
                if missing_pct > 50:
                    # Too many missing - consider dropping
                    print(f"Warning: {col} has {missing_pct:.1f}% missing values")
                    self.cleaning_report[f'{col}_missing'] =
'high_missing_rate'
```

```

        elif self.df[col].dtype in ['float64', 'int64']:
            # Numeric column - use appropriate imputation
            if missing_pct < 5:
                # Few missing - use mean
                self.df[col].fillna(self.df[col].mean(),
inplace=True)
                self.cleaning_report[f'{col}_missing'] =
'mean_imputation'
            else:
                # More missing - use median (robust to
outliers)
                self.df[col].fillna(self.df[col].median(),
inplace=True)
                self.cleaning_report[f'{col}_missing'] =
'median_imputation'

        else:
            # Categorical column
            if missing_pct < 10:
                # Use mode
                mode_value = self.df[col].mode()[0] if not
self.df[col].mode().empty else 'Unknown'
                self.df[col].fillna(mode_value, inplace=True)
                self.cleaning_report[f'{col}_missing'] =
'mode_imputation'
            else:
                # Create 'Unknown' category
                self.df[col].fillna('Unknown', inplace=True)
                self.cleaning_report[f'{col}_missing'] =
'unknown_category'

    return self

def remove_duplicates(self, subset=None, keep='first'):
    """Remove duplicate records"""
    initial_shape = self.df.shape[0]
    self.df.drop_duplicates(subset=subset, keep=keep, inplace=True)
    removed = initial_shape - self.df.shape[0]

    self.cleaning_report['duplicates_removed'] = removed
    print(f"Removed {removed} duplicate records")

    return self

def handle_outliers(self, columns=None, method='iqr', threshold=3):
    """Handle outliers using various methods"""

    if columns is None:
        columns = self.df.select_dtypes(include=[np.number]).columns

```

```

outlier_report = {}

for col in columns:
    if method == 'iqr':
        Q1 = self.df[col].quantile(0.25)
        Q3 = self.df[col].quantile(0.75)
        IQR = Q3 - Q1

        lower_bound = Q1 - threshold * IQR
        upper_bound = Q3 + threshold * IQR

        outliers = (self.df[col] < lower_bound) | (self.df[col] >
upper_bound)
        outlier_count = outliers.sum()

        if outlier_count > 0:
            # Cap outliers
            self.df.loc[self.df[col] < lower_bound, col] =
lower_bound
            self.df.loc[self.df[col] > upper_bound, col] =
upper_bound
            outlier_report[col] = {
                'method': 'iqr_capping',
                'count': outlier_count,
                'bounds': (lower_bound, upper_bound)
            }

    elif method == 'zscore':
        z_scores = np.abs(stats.zscore(self.df[col].dropna()))
        outliers = z_scores > threshold
        outlier_count = outliers.sum()

        if outlier_count > 0:
            # Remove outliers
            self.df = self.df[z_scores <= threshold]
            outlier_report[col] = {
                'method': 'zscore_removal',
                'count': outlier_count,
                'threshold': threshold
            }

self.cleaning_report['outliers'] = outlier_report
return self

def standardize_formats(self):
    """Standardize common format issues"""

    # Standardize IP addresses
    ip_pattern = r'\b(?:[0-9]{1,3}\.){3}[0-9]{1,3}\b'

```

```

        for col in self.df.columns:
            if 'ip' in col.lower() or 'address' in col.lower():
                if self.df[col].dtype == 'object':
                    # Validate IP addresses
                    valid_ips = self.df[col].str.match(ip_pattern)
                    invalid_count = (~valid_ips).sum()

                    if invalid_count > 0:
                        print(f"Found {invalid_count} invalid IPs in
{col}")
                        self.cleaning_report[f'{col}_invalid_ips'] =
invalid_count

            # Standardize timestamps
            for col in self.df.columns:
                if 'time' in col.lower() or 'date' in col.lower():
                    try:
                        self.df[col] = pd.to_datetime(self.df[col],
errors='coerce')
                        self.cleaning_report[f'{col}_datetime'] = 'converted'
                    except:
                        pass

            # Standardize text fields
            text_columns = self.df.select_dtypes(include=['object']).columns

            for col in text_columns:
                # Remove extra whitespace
                self.df[col] = self.df[col].str.strip()

            # Standardize case for categorical fields
            if self.df[col].nunique() < 50: # Likely categorical
                # Check if mostly uppercase or lowercase
                sample = self.df[col].dropna().head(100)
                if sample.str.isupper().sum() > len(sample) * 0.8:
                    continue # Keep uppercase
                elif sample.str.islower().sum() > len(sample) * 0.8:
                    continue # Keep lowercase
                else:
                    # Mixed case - standardize to title case
                    self.df[col] = self.df[col].str.title()
                    self.cleaning_report[f'{col}_case'] = 'title_case'

        return self

    def validate_data(self, rules):
        """Validate data against business rules"""

        validation_report = {}

```

```
for col, rule in rules.items():
    if col not in self.df.columns:
        continue

    if 'min' in rule and 'max' in rule:
        # Range validation
        invalid = (self.df[col] < rule['min']) | (self.df[col] >
rule['max'])
        invalid_count = invalid.sum()

        if invalid_count > 0:
            validation_report[col] = {
                'rule': f"Range [{rule['min']}, {rule['max']}]",
                'violations': invalid_count
            }

    if 'values' in rule:
        # Allowed values validation
        invalid = ~self.df[col].isin(rule['values'])
        invalid_count = invalid.sum()

        if invalid_count > 0:
            validation_report[col] = {
                'rule': f"Allowed values: {rule['values']}",
                'violations': invalid_count
            }

    if 'pattern' in rule:
        # Regex pattern validation
        if self.df[col].dtype == 'object':
            invalid = ~self.df[col].str.match(rule['pattern'])
            invalid_count = invalid.sum()

            if invalid_count > 0:
                validation_report[col] = {
                    'rule': f"Pattern: {rule['pattern']}",
                    'violations': invalid_count
                }

self.cleaning_report['validation'] = validation_report
return validation_report

def encode_categorical(self, encoding_type='label'):
    """Encode categorical variables"""

    from sklearn.preprocessing import LabelEncoder, OneHotEncoder

    categorical_columns = self.df.select_dtypes(include=
['object']).columns
```

```

encoding_report = {}

for col in categorical_columns:
    unique_count = self.df[col].nunique()

    if unique_count < 2:
        # Single value - drop column
        self.df.drop(col, axis=1, inplace=True)
        encoding_report[col] = 'dropped_single_value'

    elif unique_count == 2:
        # Binary encoding
        le = LabelEncoder()
        self.df[col] = le.fit_transform(self.df[col])
        encoding_report[col] = 'binary_encoded'

    elif unique_count < 10 and encoding_type == 'onehot':
        # One-hot encoding for low cardinality
        dummies = pd.get_dummies(self.df[col], prefix=col)
        self.df = pd.concat([self.df, dummies], axis=1)
        self.df.drop(col, axis=1, inplace=True)
        encoding_report[col] = 'one_hot_encoded'

    else:
        # Label encoding for high cardinality
        le = LabelEncoder()
        self.df[col] = le.fit_transform(self.df[col])
        encoding_report[col] = 'label_encoded'

self.cleaning_report['encoding'] = encoding_report
return self

def get_clean_data(self):
    """Return cleaned DataFrame"""
    return self.df

def get_report(self):
    """Return cleaning report"""
    return self.cleaning_report

# Example usage
def clean_security_logs():
    """Example of cleaning security log data"""

    # Load sample data
    data = {
        'timestamp': ['2024-01-15 10:30:00', '2024-01-15 10:31:00', None,
        '2024-01-15 10:33:00'],
        'source_ip': ['192.168.1.100', '192.168.1.101', '192.168.1.100',
        'invalid_ip'],

```

```

'destination_port': [80, 443, 99999, 22], # 99999 is outlier
'protocol': ['HTTP', 'HTTPS', 'http', None],
'bytes_sent': [1024, 2048, None, 512],
'severity': ['HIGH', 'medium', 'Low', 'HIGH']
}

df = pd.DataFrame(data)
print("Original Data:")
print(df)
print("\n")

# Initialize cleaner
cleaner = DataCleaner(df)

# Generate initial report
initial_report = cleaner.generate_report()
print("Initial Data Quality Report:")
print(f"Shape: {initial_report['shape']} ")
print(f"Missing values: {initial_report['missing_values']} ")
print(f"Issues: {initial_report['issues']} ")
print("\n")

# Define validation rules
validation_rules = {
    'destination_port': {'min': 1, 'max': 65535},
    'protocol': {'values': ['HTTP', 'HTTPS', 'SSH', 'FTP']},
    'source_ip': {'pattern': r'\b(?:[0-9]{1,3}\.){3}[0-9]{1,3}\b'}
}

# Clean the data
cleaned_df = (cleaner
              .handle_missing_values()
              .remove_duplicates()
              .handle_outliers(['destination_port', 'bytes_sent'])
              .standardize_formats()
              .get_clean_data())

# Validate
validation_report = cleaner.validate_data(validation_rules)

print("\nCleaned Data:")
print(cleaned_df)
print("\n")

print("Cleaning Report:")
print(cleaner.get_report())

return cleaned_df

# Advanced cleaning for time series data

```

```

def clean_time_series_security_data():
    """Clean time series security data with special handling"""

    # Generate sample time series data
    dates = pd.date_range('2024-01-01', periods=1000, freq='H')
    data = {
        'timestamp': dates,
        'event_count': np.random.poisson(10, 1000),
        'avg_severity': np.random.uniform(1, 5, 1000),
        'unique_ips': np.random.randint(5, 50, 1000)
    }

    # Add some anomalies
    data['event_count'][100:105] = 1000 # Spike
    data['event_count'][500:510] = 0     # Dropout

    df = pd.DataFrame(data)

    # Time series specific cleaning
    class TimeSeriesCleaner:
        def __init__(self, df, timestamp_col):
            self.df = df.copy()
            self.timestamp_col = timestamp_col
            self.df.set_index(timestamp_col, inplace=True)

        def detect_gaps(self, expected_freq='H'):
            """Detect gaps in time series"""
            # Check for missing timestamps
            expected_index = pd.date_range(
                start=self.df.index.min(),
                end=self.df.index.max(),
                freq=expected_freq
            )

            missing_timestamps = expected_index.difference(self.df.index)

            if len(missing_timestamps) > 0:
                print(f"Found {len(missing_timestamps)} missing
timestamps")

                # Fill missing timestamps
                self.df = self.df.reindex(expected_index)

            return self

        def handle_spikes(self, columns, method='rolling_median',
window=24):
            """Handle spikes in time series data"""

            for col in columns:
                if method == 'rolling_median':

```

```

        rolling_median = self.df[col].rolling(window=window,
center=True).median()
        rolling_std = self.df[col].rolling(window=window,
center=True).std()

        # Detect spikes
        spike_threshold = rolling_median + 3 * rolling_std
        spikes = self.df[col] > spike_threshold

        # Replace spikes with rolling median
        self.df.loc[spikes, col] = rolling_median[spikes]

        print(f"Handled {spikes.sum()} spikes in {col}")

    return self

    def interpolate_missing(self, method='linear'):
        """Interpolate missing values in time series"""

        numeric_cols = self.df.select_dtypes(include=[np.number]).columns

        for col in numeric_cols:
            missing_count = self.df[col].isnull().sum()
            if missing_count > 0:
                self.df[col] = self.df[col].interpolate(method=method)
                print(f"Interpolated {missing_count} values in {col}")

        return self

    # Clean time series
    ts_cleaner = TimeSeriesCleaner(df, 'timestamp')
    cleaned_ts = (ts_cleaner
                  .detect_gaps()
                  .handle_spikes(['event_count'])
                  .interpolate_missing()
                  .df)

    # Plot before and after
    import matplotlib.pyplot as plt

    fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(12, 8))

    # Before cleaning
    ax1.plot(df.set_index('timestamp')['event_count'], label='Original',
alpha=0.7)
    ax1.set_title('Before Cleaning')
    ax1.set_ylabel('Event Count')
    ax1.legend()

```

```
# After cleaning
ax2.plot(cleaned_ts['event_count'], label='Cleaned', alpha=0.7)
ax2.set_title('After Cleaning')
ax2.set_xlabel('Timestamp')
ax2.set_ylabel('Event Count')
ax2.legend()

plt.tight_layout()
plt.show()

return cleaned_ts

if __name__ == "__main__":
    # Clean regular data
    cleaned_df = clean_security_logs()

    # Clean time series data
    cleaned_ts = clean_time_series_security_data()
```

## **Exercise 1: Complete Data Pipeline**

Build a complete data pipeline that:

1. Scraps security advisories from a website
2. Stores raw data in MongoDB
3. Cleans and validates the data
4. Exports to SQL database for analysis
5. Visualizes key metrics

## **Exercise 2: Real-time Log Processor**

Create a system that:

1. Monitors a log file in real-time
2. Parses different log formats (Apache, nginx, syslog)
3. Detects anomalies using statistical methods
4. Stores results in appropriate databases
5. Provides real-time visualization dashboard

## **Exercise 3: Multi-Source Data Integration**

86

Combine data from:

- SQL database (user information)
- MongoDB (event logs)
- CSV files (historical data)
- Web API (real-time updates)

Requirements:

1. Handle different data formats and schemas
  2. Resolve entity matching issues
  3. Deal with temporal alignment
  4. Create a unified dataset for analysis
  5. Implement incremental updates
-

This section covered the essential data handling skills for machine learning:

## Key Concepts Learned:

- **Python & NumPy**: Efficient data structures and vectorized operations
- **SQL**: Structured queries for relational data with aggregation techniques
- **NoSQL**: Document-based storage for flexible schemas
- **Web Scraping**: Ethical data extraction from web sources
- **Data Cleaning**: Systematic approach to data quality issues
- **Visualization**: Creating effective plots with matplotlib

## Important Takeaways:

1. **Choose the right tool**: SQL for structured, MongoDB for flexible, scraping for external
2. **Data quality matters**: Garbage in, garbage out - cleaning is crucial
3. **Vectorization**: Use NumPy/Pandas operations instead of loops
4. **Ethics**: Always respect robots.txt and rate limits when scraping
5. **Documentation**: Track all transformations for reproducibility
6. **Visualization**: Always label axes and use appropriate plot types

## Next Steps:

87

In Section 2, we'll use these clean datasets to explore statistical analysis and build our first predictive models. The quality of your data preparation directly impacts model performance!

# Machine Learning Training Documentation

## Section 2: Data Exploration and Statistics

### Overview

This section builds the mathematical foundation for machine learning through statistics and probability theory. We'll explore measures of data distribution, learn probabilistic reasoning with Bayes' theorem, and apply signal processing techniques for pattern detection in security data.

### Learning Objectives

By the end of this section, you will be able to:

- Calculate and interpret statistical measures correctly
- Apply probability theory to real-world scenarios
- Use Bayes' theorem for inference and classification
- Implement Fourier analysis for periodic pattern detection
- Perform statistical hypothesis testing
- Build statistical models for anomaly detection

# Measures of Central Tendency

## Definitions and Concepts

### Mean (Average)

- **Definition:** Sum of all values divided by the count of values
- **Formula:**  $\mu = (\Sigma x) / n$
- **Characteristics:**
  - Sensitive to outliers
  - Best for symmetric distributions
  - Has unique algebraic properties

### Median

- **Definition:** Middle value when data is sorted
- **Formula:**
  - Odd n:  $x[(n+1)/2]$
  - Even n:  $(x[n/2] + x[n/2+1]) / 2$
- **Characteristics:**
  - Robust to outliers
  - Better for skewed distributions
  - 50th percentile

89

### Mode

- **Definition:** Most frequently occurring value(s)
- **Characteristics:**
  - Can have multiple modes (multimodal)
  - Only measure for categorical data
  - May not exist (all values unique)

## Comparison Table

| Measure | Best Used When                    | Sensitive to Outliers | Example Use Case        |
|---------|-----------------------------------|-----------------------|-------------------------|
| Mean    | Data is symmetric, no outliers    | Yes                   | Average response time   |
| Median  | Data is skewed or has outliers    | No                    | Typical salary          |
| Mode    | Categorical data or finding peaks | No                    | Most common attack type |

# Visual Memory Aid

Dataset: [1, 2, 2, 3, 100]

Mean =  $(1+2+2+3+100)/5 = 21.6$  ← Pulled up by outlier

Median = 2 ← Stays centered

Mode = 2 ← Most frequent

## Measures of Spread (Variability)

### Core Concepts

#### Variance ( $\sigma^2$ )

- **Definition:** Average squared deviation from the mean
- **Population formula:**  $\sigma^2 = \sum(x_i - \mu)^2 / N$
- **Sample formula:**  $s^2 = \sum(x_i - \bar{x})^2 / (n-1)$
- **Why n-1?**: Bessel's correction for sample bias

#### Standard Deviation ( $\sigma$ )

- **Definition:** Square root of variance
- **Formula:**  $\sigma = \sqrt{\text{variance}}$
- **Interpretation:** Average distance from mean
- **68-95-99.7 Rule:** For normal distribution
  - 68% within  $\pm 1\sigma$
  - 95% within  $\pm 2\sigma$
  - 99.7% within  $\pm 3\sigma$

90

#### Coefficient of Variation (CV)

- **Formula:**  $CV = (\sigma / \mu) \times 100\%$
- **Use:** Compare variation between different scales
- **Example:** Compare packet size variation (bytes) vs response time variation (ms)

## Robust Measures of Spread

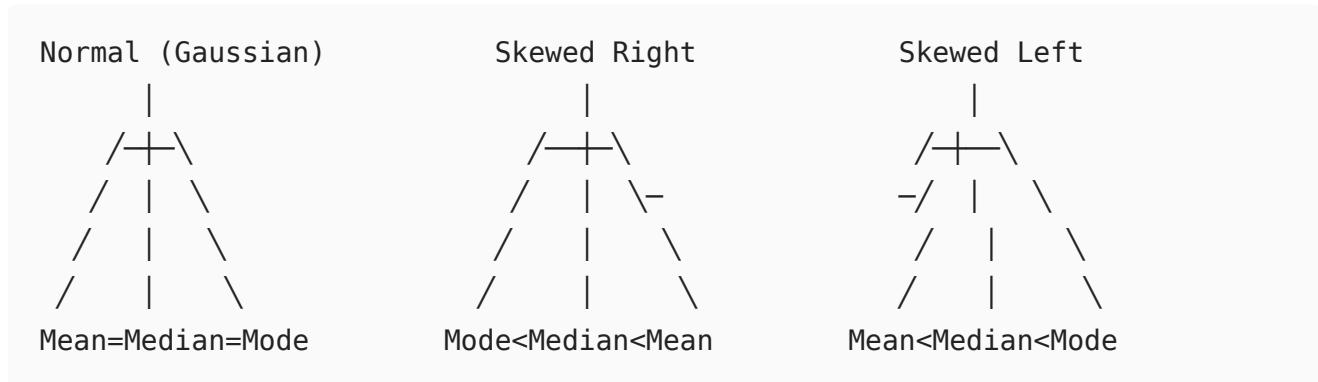
| Measure     | Formula               | Breakdown Point | Use When          |
|-------------|-----------------------|-----------------|-------------------|
| Range       | Max - Min             | 0%              | Quick assessment  |
| IQR         | Q3 - Q1               | 25%             | Outliers present  |
| MAD         | Median( xi - median ) | 50%             | Heavy outliers    |
| Trimmed Std | Std of middle 80%     | Variable        | Contaminated data |

## Memory Tricks

- **Variance:** "Average of squares minus square of average"
- **Standard Deviation:** "Typical distance from typical value"
- **IQR:** "The middle 50% spread"
- **MAD:** "Median Absolute Deviation - the robust cousin of std dev"

## Distribution Shapes

### Key Distribution Characteristics



## Skewness and Kurtosis

### Skewness (Asymmetry)

91

- **Formula:**  $\gamma_1 = E[(X-\mu)^3] / \sigma^3$
- **Interpretation:**
  - $\gamma_1 = 0$ : Symmetric
  - $\gamma_1 > 0$ : Right-skewed (tail to right)
  - $\gamma_1 < 0$ : Left-skewed (tail to left)

### Kurtosis (Tail heaviness)

- **Formula:**  $\gamma_2 = E[(X-\mu)^4] / \sigma^4 - 3$
- **Interpretation:**
  - $\gamma_2 = 0$ : Normal tails (mesokurtic)
  - $\gamma_2 > 0$ : Heavy tails (leptokurtic)
  - $\gamma_2 < 0$ : Light tails (platykurtic)

## Practical Implementation

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from scipy import stats
```

```

# Generate different types of data for demonstration
np.random.seed(42)

# Normal distribution
normal_data = np.random.normal(100, 15, 1000)

# Skewed distribution (log-normal - common in network traffic)
skewed_data = np.random.lognormal(4, 0.5, 1000)

# Data with outliers (mix of normal and extreme values)
outlier_data = np.concatenate([
    np.random.normal(50, 5, 950),
    np.random.normal(150, 5, 50) # 5% outliers
])

# Create comprehensive statistical analysis
class StatisticalAnalyzer:
    """Comprehensive statistical analysis toolkit"""

    def __init__(self, data, name="Dataset"):
        self.data = np.array(data)
        self.name = name
        self.stats = {}

    def calculate_central_measures(self):
        """Calculate measures of central tendency"""
        self.stats['mean'] = np.mean(self.data)
        self.stats['median'] = np.median(self.data)
        self.stats['mode'] = stats.mode(self.data, keepdims=True).mode[0]

        # Trimmed mean (robust)
        self.stats['trimmed_mean_10'] = stats.trim_mean(self.data, 0.1)

        # Geometric mean (for ratios)
        if np.all(self.data > 0):
            self.stats['geometric_mean'] = stats.gmean(self.data)

    return self.stats

    def calculate_spread_measures(self):
        """Calculate measures of spread"""
        self.stats['variance'] = np.var(self.data)
        self.stats['std_dev'] = np.std(self.data)
        self.stats['cv'] = (self.stats['std_dev'] / self.stats['mean']) * 100

        # Robust measures
        self.stats['iqr'] = stats.iqr(self.data)
        self.stats['mad'] = stats.median_abs_deviation(self.data)

```

```

# Percentiles
self.stats['percentiles'] = {
    '1%': np.percentile(self.data, 1),
    '5%': np.percentile(self.data, 5),
    '25%': np.percentile(self.data, 25),
    '50%': np.percentile(self.data, 50),
    '75%': np.percentile(self.data, 75),
    '95%': np.percentile(self.data, 95),
    '99%': np.percentile(self.data, 99)
}

return self.stats

def calculate_shape_measures(self):
    """Calculate distribution shape measures"""
    self.stats['skewness'] = stats.skew(self.data)
    self.stats['kurtosis'] = stats.kurtosis(self.data)

    # Test for normality
    statistic, p_value = stats.normaltest(self.data)
    self.stats['normality_test'] = {
        'statistic': statistic,
        'p_value': p_value,
        'is_normal': p_value > 0.05
    }

    return self.stats

def detect_outliers_multiple_methods(self):
    """Detect outliers using multiple methods"""
    outliers = {}

    # Method 1: Z-score
    z_scores = np.abs(stats.zscore(self.data))
    outliers['z_score'] = self.data[z_scores > 3]

    # Method 2: IQR
    Q1 = np.percentile(self.data, 25)
    Q3 = np.percentile(self.data, 75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    outliers['iqr'] = self.data[(self.data < lower_bound) | (self.data > upper_bound)]

    # Method 3: MAD
    median = np.median(self.data)
    mad = stats.median_abs_deviation(self.data)
    modified_z_scores = 0.6745 * (self.data - median) / mad

```

```

        outliers['mad'] = self.data[np.abs(modified_z_scores) > 3.5]

        self.stats['outliers'] = {
            'z_score_count': len(outliers['z_score']),
            'iqr_count': len(outliers['iqr']),
            'mad_count': len(outliers['mad'])
        }

    }

    return outliers

def visualize_distribution(self):
    """Create comprehensive distribution visualization"""
    fig, axes = plt.subplots(2, 2, figsize=(12, 10))

    # Histogram with KDE
    ax1 = axes[0, 0]
    ax1.hist(self.data, bins=30, density=True, alpha=0.7,
edgecolor='black')
    kde = stats.gaussian_kde(self.data)
    x_range = np.linspace(self.data.min(), self.data.max(), 100)
    ax1.plot(x_range, kde(x_range), 'r-', linewidth=2, label='KDE')
    ax1.axvline(self.stats['mean'], color='red', linestyle='--',
label=f'Mean: {self.stats["mean"]:.2f}')
    ax1.axvline(self.stats['median'], color='green', linestyle='--',
label=f'Median: {self.stats["median"]:.2f}')
    ax1.set_title(f'{self.name} - Distribution')
    ax1.legend()

    # Box plot
    ax2 = axes[0, 1]
    box_data = ax2.boxplot(self.data, vert=True, patch_artist=True)
    box_data['boxes'][0].set_facecolor('lightblue')
    ax2.set_title(f'{self.name} - Box Plot')
    ax2.set_ylabel('Value')

    # Q-Q plot
    ax3 = axes[1, 0]
    stats.probplot(self.data, dist="norm", plot=ax3)
    ax3.set_title(f'{self.name} - Q-Q Plot')

    # Statistics summary
    ax4 = axes[1, 1]
    ax4.axis('off')
    summary_text = f"""

Statistical Summary for {self.name}:

Central Tendency:
Mean: {self.stats['mean']:.2f}
Median: {self.stats['median']:.2f}
Trimmed Mean (10%): {self.stats['trimmed_mean_10']:.2f}
"""

```

```

Spread:
Std Dev: {self.stats['std_dev']:.2f}
IQR: {self.stats['iqr']:.2f}
MAD: {self.stats['mad']:.2f}
CV: {self.stats['cv']:.2f}%

Shape:
Skewness: {self.stats['skewness']:.2f}
Kurtosis: {self.stats['kurtosis']:.2f}
Normal: {self.stats['normality_test']['is_normal']}

Outliers:
Z-score method: {self.stats['outliers']['z_score_count']}
IQR method: {self.stats['outliers']['iqr_count']}
MAD method: {self.stats['outliers']['mad_count']}
"""

ax4.text(0.1, 0.9, summary_text, transform=ax4.transAxes,
         fontsize=10, verticalalignment='top',
         fontfamily='monospace')

plt.tight_layout()
plt.show()

def generate_report(self):
    """Generate complete statistical report"""
    self.calculate_central_measures()
    self.calculate_spread_measures()
    self.calculate_shape_measures()
    self.detect_outliers_multiple_methods()
    self.visualize_distribution()

    return self.stats

# Analyze different distributions
print("== Analyzing Different Data Distributions ==\n")

# Normal distribution analysis
analyzer_normal = StatisticalAnalyzer(normal_data, "Normal Distribution")
stats_normal = analyzer_normal.generate_report()

# Skewed distribution analysis
analyzer_skewed = StatisticalAnalyzer(skewed_data, "Skewed Distribution")
stats_skewed = analyzer_skewed.generate_report()

# Outlier distribution analysis
analyzer_outlier = StatisticalAnalyzer(outlier_data, "Data with Outliers")
stats_outlier = analyzer_outlier.generate_report()

# Comparison of robustness

```

```
comparison_df = pd.DataFrame({  
    'Normal': [stats_normal['mean'], stats_normal['median'],  
               stats_normal['trimmed_mean_10'], stats_normal['mad']],  
    'Skewed': [stats_skewed['mean'], stats_skewed['median'],  
               stats_skewed['trimmed_mean_10'], stats_skewed['mad']],  
    'Outliers': [stats_outlier['mean'], stats_outlier['median'],  
                 stats_outlier['trimmed_mean_10'], stats_outlier['mad']]  
}, index=['Mean', 'Median', 'Trimmed Mean', 'MAD'])  
  
print("\n==== Robustness Comparison ===")  
print(comparison_df)
```

---

# Fundamental Concepts

## Basic Probability Definitions

### Sample Space (S)

- Definition: Set of all possible outcomes
- Example: For a die roll,  $S = \{1, 2, 3, 4, 5, 6\}$

### Event (E)

- Definition: Subset of sample space
- Example: "Even number" =  $\{2, 4, 6\}$

### Probability Axioms

1.  $0 \leq P(E) \leq 1$  for any event E
2.  $P(S) = 1$  (certainty)
3. For mutually exclusive events:  $P(A \cup B) = P(A) + P(B)$

## Types of Probability

97

| Type        | Definition                  | Formula                                   | Example                            |
|-------------|-----------------------------|---|------------------------------------|
| Marginal    | Probability of single event | $P(A)$                                    | $P(\text{Attack}) = 0.3$           |
| Joint       | Probability of intersection | $P(A \cap B)$                             | $P(\text{Attack AND Night}) = 0.2$ |
| Conditional | Probability given condition | $P(A B) = P(A \cap B)/P(B)$               | $P(\text{Attack Night}) = 0.6$     |
| Union       | Probability of either/or    | $P(A \cup B) = P(A) + P(B) - P(A \cap B)$ | $P(\text{Attack OR Error}) = 0.4$  |

## Probability Rules and Formulas

### Fundamental Rules

#### Complement Rule

- $P(A') = 1 - P(A)$
- "Probability of NOT A"

#### Multiplication Rule

- Independent:  $P(A \cap B) = P(A) \times P(B)$
- Dependent:  $P(A \cap B) = P(A) \times P(B|A)$

## Addition Rule

- General:  $P(A \cup B) = P(A) + P(B) - P(A \cap B)$
- Mutually exclusive:  $P(A \cup B) = P(A) + P(B)$

## Independence vs Dependence

Independent Events:

$$P(A|B) = P(A)$$

Dependent Events:

$$P(A|B) \neq P(A)$$

Example:

Coin flips

Network events on  
different servers

Example:

Drawing cards without replacement

Cascading failures

## Bayes' Theorem

### The Formula

98

**Bayes' Theorem:**  $P(A|B) = P(B|A) \times P(A) / P(B)$

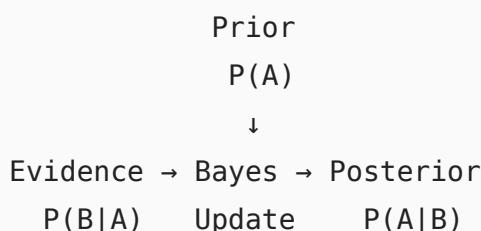
Where:

- $P(A|B)$  = Posterior probability
- $P(B|A)$  = Likelihood
- $P(A)$  = Prior probability
- $P(B)$  = Evidence

### Expanded Form

$$P(A|B) = P(B|A) \times P(A) / [P(B|A) \times P(A) + P(B|A') \times P(A')]$$

### Visual Intuition



## Common Probability Distributions

## Discrete Distributions

| Distribution | Parameters | Mean      | Variance    | Use Case                 |
|--------------|------------|-----------|-------------|--------------------------|
| Bernoulli    | p          | p         | p(1-p)      | Single yes/no event      |
| Binomial     | n, p       | np        | np(1-p)     | Count of successes       |
| Poisson      | $\lambda$  | $\lambda$ | $\lambda$   | Rare events in time      |
| Geometric    | p          | 1/p       | $(1-p)/p^2$ | Time until first success |

## Continuous Distributions

| Distribution | Parameters    | Mean                     | Variance      | Use Case             |
|--------------|---------------|--------------------------|---------------|----------------------|
| Normal       | $\mu, \sigma$ | $\mu$                    | $\sigma^2$    | Natural phenomena    |
| Exponential  | $\lambda$     | $1/\lambda$              | $1/\lambda^2$ | Time between events  |
| Log-normal   | $\mu, \sigma$ | $\exp(\mu + \sigma^2/2)$ | Complex       | Skewed positive data |
| Uniform      | a, b          | $(a+b)/2$                | $(b-a)^2/12$  | Equal likelihood     |

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy import stats
import seaborn as sns

# Probability calculations for security scenarios
class SecurityProbabilityAnalyzer:
    """Analyze probabilities in security contexts"""

    def __init__(self):
        self.events = {}
        self.conditional_probs = {}

    def demonstrate_basic_probability(self):
        """Show basic probability concepts with security examples"""

        # Sample data: Security events over 30 days
        np.random.seed(42)
        n_days = 30

        # Generate events
        events_data = {
            'day': range(1, n_days + 1),
            'malware': np.random.binomial(1, 0.4, n_days),
            'phishing': np.random.binomial(1, 0.3, n_days),
            'ddos': np.random.binomial(1, 0.1, n_days),
            'intrusion': np.random.binomial(1, 0.2, n_days)
        }

        df_events = pd.DataFrame(events_data)

        # Calculate basic probabilities
        print("== Basic Probability Calculations ==")
        for event in ['malware', 'phishing', 'ddos', 'intrusion']:
            p = df_events[event].mean()
            self.events[event] = p
            print(f"P({event}) = {p:.3f}")

        # Joint probabilities
        print("\n== Joint Probabilities ==")
        p_malware_and_phishing = ((df_events['malware'] == 1) &
                                  (df_events['phishing'] == 1)).mean()
        print(f"P(malware AND phishing) = {p_malware_and_phishing:.3f}")

        # Conditional probabilities
        print("\n== Conditional Probabilities ==")
        malware_days = df_events[df_events['malware'] == 1]
```

```

        p_phishing_given_malware = malware_days['phishing'].mean()
        print(f"P(phishing | malware) = {p_phishing_given_malware:.3f}")

    # Independence check
    expected_joint = self.events['malware'] * self.events['phishing']
    print(f"\nIndependence check:")
    print(f"Expected P(malware AND phishing) if independent:
{expected_joint:.3f}")
    print(f"Actual P(malware AND phishing):
{p_malware_and_phishing:.3f}")
    print(f"Are they independent? {np.isclose(expected_joint,
p_malware_and_phishing)}")

    return df_events

def bayes_theorem_security(self):
    """Apply Bayes' theorem to security scenarios"""

    print("\n==== Bayes' Theorem: Intrusion Detection System ====")

    # Given information
    p_intrusion = 0.001 # Prior: 0.1% of connections are intrusions
    p_alert_given_intrusion = 0.99 # Sensitivity: 99% detection rate
    p_alert_given_normal = 0.02 # False positive rate: 2%
    101
    # Calculate P(alert) using law of total probability
    p_normal = 1 - p_intrusion
    p_alert = (p_alert_given_intrusion * p_intrusion +
               p_alert_given_normal * p_normal)

    # Apply Bayes' theorem
    p_intrusion_given_alert = (p_alert_given_intrusion * p_intrusion)
    / p_alert

    print(f"Prior P(intrusion) = {p_intrusion:.4f}")
    print(f"P(alert | intrusion) = {p_alert_given_intrusion:.4f}")
    print(f"P(alert | normal) = {p_alert_given_normal:.4f}")
    print(f"P(alert) = {p_alert:.4f}")
    print(f"\nPosterior P(intrusion | alert) =
{p_intrusion_given_alert:.4f}")
    print(f"This is a {p_intrusion_given_alert/p_intrusion:.1f}x
increase from prior")

    # Visualize Bayes update
    self.visualize_bayes_update(p_intrusion, p_intrusion_given_alert)

    return p_intrusion_given_alert

def visualize_bayes_update(self, prior, posterior):
    """Visualize how Bayes' theorem updates beliefs"""

```

```

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

# Before and after
categories = ['Intrusion', 'Normal']

# Prior
prior_probs = [prior, 1-prior]
ax1.bar(categories, prior_probs, color=['red', 'green'],
alpha=0.6)
ax1.set_ylim(0, 1)
ax1.set_title('Prior Probability')
ax1.set_ylabel('Probability')
for i, v in enumerate(prior_probs):
    ax1.text(i, v + 0.01, f'{v:.4f}', ha='center')

# Posterior
posterior_probs = [posterior, 1-posterior]
ax2.bar(categories, posterior_probs, color=['red', 'green'],
alpha=0.6)
ax2.set_ylim(0, 1)
ax2.set_title('Posterior Probability (After Alert)')
ax2.set_ylabel('Probability')
for i, v in enumerate(posterior_probs):
    ax2.text(i, v + 0.01, f'{v:.4f}', ha='center')

plt.tight_layout()
plt.show()

def probability_distributions_demo(self):
    """Demonstrate common probability distributions in security
context"""

    fig, axes = plt.subplots(2, 3, figsize=(15, 10))
    axes = axes.ravel()

    # 1. Poisson: Number of attacks per hour
    lambda_attacks = 2.5
    x_poisson = np.arange(0, 10)
    pmf_poisson = stats.poisson.pmf(x_poisson, lambda_attacks)

    axes[0].bar(x_poisson, pmf_poisson, alpha=0.7)
    axes[0].set_title(f'Poisson Distribution (\u03bb={lambda_attacks})\nAttacks per Hour')
    axes[0].set_xlabel('Number of Attacks')
    axes[0].set_ylabel('Probability')

    # 2. Exponential: Time between attacks
    lambda_time = 1/30 # Average 30 minutes between attacks
    x_exp = np.linspace(0, 150, 1000)

```

```

pdf_exp = stats.expon.pdf(x_exp, scale=1/lambda_time)

axes[1].plot(x_exp, pdf_exp, linewidth=2)
axes[1].fill_between(x_exp, pdf_exp, alpha=0.3)
axes[1].set_title('Exponential Distribution\nTime Between Attacks
(minutes)')
axes[1].set_xlabel('Time (minutes)')
axes[1].set_ylabel('Probability Density')

# 3. Binomial: Success rate of phishing emails
n_emails = 100
p_success = 0.03 # 3% success rate
x_binom = np.arange(0, 15)
pmf_binom = stats.binom.pmf(x_binom, n_emails, p_success)

axes[2].bar(x_binom, pmf_binom, alpha=0.7)
axes[2].set_title(f'Binomial Distribution (n={n_emails}, p=
{p_success})\nPhishing Successes')
axes[2].set_xlabel('Number of Successes')
axes[2].set_ylabel('Probability')

# 4. Normal: Log file sizes
mu_size = 100 # MB
sigma_size = 20
x_normal = np.linspace(40, 160, 1000)
pdf_normal = stats.norm.pdf(x_normal, mu_size, sigma_size) 103

axes[3].plot(x_normal, pdf_normal, linewidth=2)
axes[3].fill_between(x_normal, pdf_normal, alpha=0.3)
axes[3].axvline(mu_size, color='red', linestyle='--',
label='Mean')
axes[3].set_title(f'Normal Distribution ( $\mu$ ={mu_size},  $\sigma$ =
{sigma_size})\nLog File Sizes (MB)')
axes[3].set_xlabel('Size (MB)')
axes[3].set_ylabel('Probability Density')
axes[3].legend()

# 5. Log-normal: Network traffic volume
mu_log = 3
sigma_log = 0.5
x_lognorm = np.linspace(0, 100, 1000)
pdf_lognorm = stats.lognorm.pdf(x_lognorm, sigma_log,
scale=np.exp(mu_log))

axes[4].plot(x_lognorm, pdf_lognorm, linewidth=2)
axes[4].fill_between(x_lognorm, pdf_lognorm, alpha=0.3)
axes[4].set_title('Log-Normal Distribution\nNetwork Traffic
(GB/hour)')
axes[4].set_xlabel('Traffic (GB/hour)')
axes[4].set_ylabel('Probability Density')

```

```

# 6. Geometric: Login attempts until success
p_login = 0.1 # 10% chance of success per attempt
x_geom = np.arange(1, 20)
pmf_geom = stats.geom.pmf(x_geom, p_login)

axes[5].bar(x_geom, pmf_geom, alpha=0.7)
axes[5].set_title(f'Geometric Distribution (p={p_login})\nLogin Attempts Until Success')
axes[5].set_xlabel('Number of Attempts')
axes[5].set_ylabel('Probability')

plt.tight_layout()
plt.show()

# Build Naive Bayes Classifier from scratch
class NaiveBayesSecurityClassifier:
    """Naive Bayes classifier for security event classification"""

    def __init__(self):
        self.class_priors = {}
        self.feature_likelihoods = {}
        self.classes = []
        self.features = []

    def fit(self, X, y):
        """Train the Naive Bayes classifier"""
        self.classes = np.unique(y)
        self.features = X.columns.tolist()

        # Calculate class priors
        for c in self.classes:
            self.class_priors[c] = (y == c).mean()

        # Calculate feature likelihoods
        self.feature_likelihoods = {}
        for c in self.classes:
            self.feature_likelihoods[c] = {}
            class_data = X[y == c]

            for feature in self.features:
                # For continuous features, assume Gaussian
                if X[feature].dtype in ['float64', 'int64']:
                    mean = class_data[feature].mean()
                    std = class_data[feature].std()
                    self.feature_likelihoods[c][feature] = {
                        'type': 'gaussian',
                        'mean': mean,
                        'std': std
                    }

```

```

    else:
        # For categorical features
        value_counts = class_data[feature].value_counts()
        total = len(class_data)
        probs = {}
        for value in X[feature].unique():
            # Laplace smoothing
            count = value_counts.get(value, 0) + 1
            probs[value] = count / (total +
len(X[feature].unique())))
        self.feature_likelihoods[c][feature] = {
            'type': 'categorical',
            'probs': probs
        }

    def predict_proba(self, X):
        """Predict probabilities for each class"""
        results = []

        for _, row in X.iterrows():
            probs = {}

            for c in self.classes:
                # Start with class prior (log probability)
                log_prob = np.log(self.class_priors[c])

                # Multiply by feature likelihoods
                for feature in self.features:
                    value = row[feature]
                    likelihood_info = self.feature_likelihoods[c][feature]

                    if likelihood_info['type'] == 'gaussian':
                        # Gaussian likelihood
                        mean = likelihood_info['mean']
                        std = likelihood_info['std']
                        if std > 0:
                            likelihood = stats.norm.pdf(value, mean, std)
                        else:
                            likelihood = 1.0 if value == mean else 0.0001
                    else:
                        # Categorical likelihood
                        likelihood = likelihood_info['probs'].get(value,
0.0001)

                    log_prob += np.log(likelihood)

                probs[c] = log_prob

            # Convert from log probabilities and normalize
            max_log_prob = max(probs.values())

```

```

        for c in self.classes:
            probs[c] = np.exp(probs[c] - max_log_prob)

        total = sum(probs.values())
        for c in self.classes:
            probs[c] /= total

        results.append(probs)

    return pd.DataFrame(results)

def predict(self, X):
    """Predict most likely class"""
    proba = self.predict_proba(X)
    return proba.idxmax(axis=1)

# Demonstrate probability concepts
analyzer = SecurityProbabilityAnalyzer()

# Basic probability
events_df = analyzer.demonstrate_basic_probability()

# Bayes' theorem
posterior = analyzer.bayes_theorem_security()

# Probability distributions
analyzer.probability_distributions_demo()

# Naive Bayes example
print("\n==== Naive Bayes Classifier Example ====")

# Create sample security event data
np.random.seed(42)
n_samples = 1000

# Features
packet_size = np.concatenate([
    np.random.normal(1500, 200, 700), # Normal
    np.random.normal(100, 50, 300) # Attack
])

packet_rate = np.concatenate([
    np.random.normal(100, 20, 700), # Normal
    np.random.normal(1000, 200, 300) # Attack
])

time_of_day = np.concatenate([
    np.random.choice(['morning', 'afternoon', 'evening', 'night'], 700),
    np.random.choice(['night', 'evening'], 300, p=[0.7, 0.3])
])

```

```

# Labels
labels = np.array(['normal'] * 700 + ['attack'] * 300)

# Create DataFrame
security_data = pd.DataFrame({
    'packet_size': packet_size,
    'packet_rate': packet_rate,
    'time_of_day': time_of_day,
    'label': labels
})

# Split data
from sklearn.model_selection import train_test_split
X = security_data.drop('label', axis=1)
y = security_data['label']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

# Train classifier
nb_classifier = NaiveBayesSecurityClassifier()
nb_classifier.fit(X_train, y_train)

# Make predictions
predictions = nb_classifier.predict(X_test)
probabilities = nb_classifier.predict_proba(X_test)

# Evaluate
accuracy = (predictions == y_test).mean()
print(f"Accuracy: {accuracy:.3f} ")

# Show example predictions
print("\n==== Example Predictions ===")
sample_indices = np.random.choice(len(X_test), 5)
for idx in sample_indices:
    actual_idx = X_test.index[idx]
    print(f"\nSample {idx}:")
    print(f"  Features: {X_test.iloc[idx].to_dict()}")
    print(f"  True label: {y_test.iloc[idx]}")
    print(f"  Predicted: {predictions.iloc[idx]}")
    print(f"  Probabilities: Normal={probabilities.iloc[idx]['normal']:.3f}, "
          f"Attack={probabilities.iloc[idx]['attack']:.3f}")

```

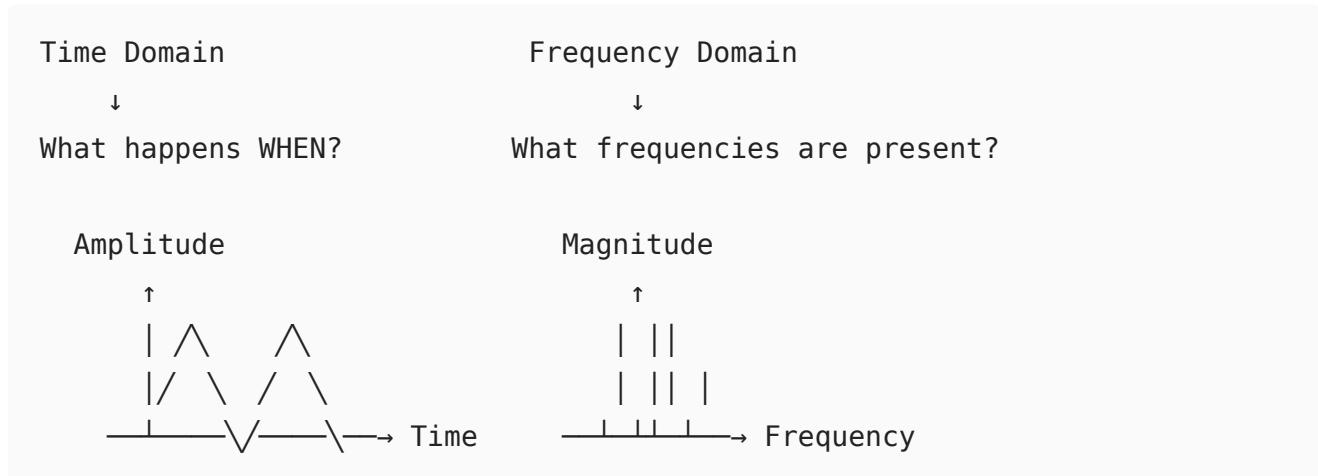
# Understanding Signals and Frequencies

## What is a Signal?

A signal is any quantity that varies with time, space, or another independent variable. In security:

- **Network traffic:** Packets/second over time
- **CPU usage:** Utilization percentage over time
- **Log events:** Event frequency over time

## Time Domain vs Frequency Domain



## Fourier Series and Transform

### Mathematical Foundation

**Fourier's Insight:** Any periodic signal can be decomposed into a sum of simple sine and cosine waves.

**Fourier Series** (for periodic signals):

$$f(t) = a_0/2 + \sum [a_n \cos(n\omega t) + b_n \sin(n\omega t)]$$

Where:

- $a_0$  = DC component (average)
- $a_n, b_n$  = Fourier coefficients
- $\omega = 2\pi/T$  (fundamental frequency)

**Fourier Transform** (for non-periodic signals):

$$F(\omega) = \int f(t) e^{-j\omega t} dt$$

**Discrete Fourier Transform (DFT):**

$$X[k] = \sum x[n] e^{-j2\pi kn/N}$$

## Key Properties

| Property    | Time Domain     | Frequency Domain             |
|-------------|-----------------|------------------------------|
| Linearity   | $af(t) + bg(t)$ | $aF(\omega) + bG(\omega)$    |
| Time shift  | $f(t - t_0)$    | $F(\omega)e^{-j\omega t_0}$  |
| Scaling     | $f(at)$         | $(1/ a )F(\omega/a)$         |
| Convolution | $f(t) * g(t)$   | $F(\omega) \times G(\omega)$ |

# Algorithm Efficiency

- **DFT:**  $O(N^2)$  operations
- **FFT:**  $O(N \log N)$  operations
- Requirement:  $N$  should be power of 2 for optimal performance

# Nyquist-Shannon Sampling Theorem

**Key Principle:** To accurately represent a signal, sample at least twice the highest frequency present.

- Nyquist frequency = Sampling rate / 2
- Aliasing occurs when sampling below Nyquist rate

# Practical Applications in Security

## 1. Beaconing Detection

Malware often communicates with C&C servers at regular intervals.

## 2. DDoS Pattern Recognition

Attack traffic often shows periodic patterns.

110

## 3. Anomaly Detection

Normal behavior has characteristic frequency signatures.

## 4. Data Exfiltration

Regular data transfers at unusual times.

# Signal Analysis Implementation

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.fft import fft, fftfreq, ifft
from scipy import signal
import pandas as pd
from datetime import datetime, timedelta

class SecuritySignalAnalyzer:
    """Signal analysis toolkit for security applications"""

    def __init__(self, sampling_rate=1.0):
        self.sampling_rate = sampling_rate
        self.nyquist_freq = sampling_rate / 2
```

```

def generate_security_signals(self):
    """Generate synthetic security signals for demonstration"""

    # Time array (1 week of hourly data)
    duration = 168 # hours
    t = np.arange(0, duration, 1) # hourly samples

    # 1. Normal traffic pattern (daily + weekly cycles)
    daily_pattern = 1000 + 500 * np.sin(2 * np.pi * t / 24) # 24-hour
cycle
    weekly_pattern = 200 * np.sin(2 * np.pi * t / 168) # Weekly cycle
    noise = np.random.normal(0, 100, len(t))
    normal_traffic = daily_pattern + weekly_pattern + noise

    # 2. Add hidden beaconing (malware communication every 6 hours)
    beacon_freq = 1/6 # cycles per hour
    beacon_signal = 300 * np.sin(2 * np.pi * beacon_freq * t)

    # 3. Add DDoS attack pattern (starts at hour 72)
    ddos_signal = np.zeros_like(t)
    ddos_start = 72
    ddos_signal[ddos_start:] = 2000 * np.sin(2 * np.pi * 0.5 *
t[ddos_start:])

    # Combine signals
    total_signal = normal_traffic + beacon_signal + ddos_signal

    return t, normal_traffic, beacon_signal, ddos_signal, total_signal

def analyze_frequency_spectrum(self, signal, signal_name="Signal"):
    """Perform FFT and analyze frequency components"""

    # Compute FFT
    N = len(signal)
    yf = fft(signal)
    xf = fftfreq(N, 1/self.sampling_rate)

    # Compute power spectral density
    psd = (2/N) * np.abs(yf[:N//2])**2
    frequencies = xf[:N//2]

    # Find dominant frequencies
    threshold = np.mean(psd) + 2 * np.std(psd)
    peaks, properties = signal.find_peaks(psd, height=threshold)
    dominant_freqs = frequencies[peaks]
    dominant_periods = 1 / dominant_freqs[dominant_freqs > 0]

    # Visualize results
    fig, axes = plt.subplots(3, 1, figsize=(12, 10))

```

```

# Time domain
ax1 = axes[0]
time_hours = np.arange(len(signal))
ax1.plot(time_hours, signal)
ax1.set_xlabel('Time (hours)')
ax1.set_ylabel('Traffic (requests/hour)')
ax1.set_title(f'{signal_name} - Time Domain')
ax1.grid(True, alpha=0.3)

# Frequency domain (full spectrum)
ax2 = axes[1]
ax2.semilogy(frequencies, psd)
ax2.set_xlabel('Frequency (cycles/hour)')
ax2.set_ylabel('Power Spectral Density')
ax2.set_title(f'{signal_name} - Frequency Domain')
ax2.grid(True, alpha=0.3)
ax2.set_xlim(0, 0.5) # Up to Nyquist frequency

# Mark detected peaks
for freq, period in zip(dominant_freqs[dominant_freqs > 0],
                         dominant_periods[dominant_periods < 200]):
    ax2.axvline(freq, color='red', linestyle='--', alpha=0.5)
    ax2.text(freq, ax2.get_ylim()[1]*0.5, f'{period:.1f}h',
              rotation=90, ha='right')

# Zoomed frequency domain
ax3 = axes[2]
ax3.plot(frequencies, psd)
ax3.set_xlabel('Frequency (cycles/hour)')
ax3.set_ylabel('Power Spectral Density')
ax3.set_title(f'{signal_name} - Frequency Domain (Zoomed)')
ax3.grid(True, alpha=0.3)
ax3.set_xlim(0, 0.2) # Focus on lower frequencies

plt.tight_layout()
plt.show()

return frequencies, psd, dominant_freqs, dominant_periods

def detect_periodic_anomalies(self, signal, window_size=24,
slide_step=6):
    """Detect periodic anomalies using sliding window FFT"""

    anomalies = []
    anomaly_scores = []
    time_windows = []

    # Baseline: first window
    baseline_window = signal[:window_size]

```

```

baseline_fft = np.abs(fft(baseline_window))
baseline_profile = baseline_fft[:window_size//2]

# Slide window through signal
for start in range(0, len(signal) - window_size, slide_step):
    end = start + window_size
    window = signal[start:end]

    # Compute FFT for window
    window_fft = np.abs(fft(window))
    window_profile = window_fft[:window_size//2]

    # Compare with baseline using various metrics
    # 1. Euclidean distance
    euclidean_dist = np.linalg.norm(window_profile -
baseline_profile)

    # 2. Correlation
    correlation = np.corrcoef(window_profile, baseline_profile)[0,
1]

    # 3. New frequency detection
    window_peaks = signal.find_peaks(window_profile,
                                      height=np.mean(window_profile)
+ 2*np.std(window_profile))[0]
    baseline_peaks = signal.find_peaks(baseline_profile,
height=np.mean(baseline_profile) + 2*np.std(baseline_profile))[0]
    new_peaks = set(window_peaks) - set(baseline_peaks)

    # Anomaly score (combine metrics)
    anomaly_score = euclidean_dist * (1 - correlation) * (1 +
len(new_peaks))

    anomaly_scores.append(anomaly_score)
    time_windows.append((start, end))

    # Detect anomaly if score exceeds threshold
    if anomaly_score > np.mean(anomaly_scores) + 2 *
np.std(anomaly_scores):
        anomalies.append({
            'start': start,
            'end': end,
            'score': anomaly_score,
            'new_frequencies': new_peaks
        })

return anomalies, anomaly_scores, time_windows

def demonstrate_aliasing(self):

```

```

"""Demonstrate the importance of proper sampling"""

fig, axes = plt.subplots(2, 2, figsize=(12, 8))
axes = axes.ravel()

# True signal: 10 Hz
true_freq = 10
t_dense = np.linspace(0, 1, 1000)
true_signal = np.sin(2 * np.pi * true_freq * t_dense)

# Different sampling rates
sampling_rates = [5, 15, 25, 100] # Hz

for idx, fs in enumerate(sampling_rates):
    ax = axes[idx]

    # Sample the signal
    t_samples = np.arange(0, 1, 1/fs)
    samples = np.sin(2 * np.pi * true_freq * t_samples)

    # Plot true signal and samples
    ax.plot(t_dense, true_signal, 'b-', alpha=0.3, label='True
signal (10 Hz)')
    ax.plot(t_samples, samples, 'ro-', label=f'Samples at {fs}
Hz')

    # Add Nyquist frequency line
    nyquist = fs / 2
    if nyquist < true_freq:
        ax.set_title(f'Sampling at {fs} Hz - ALIASING! (Nyquist=
{nyquist} Hz)')
        ax.patch.set_facecolor('#ffeeee')
    else:
        ax.set_title(f'Sampling at {fs} Hz - OK (Nyquist={nyquist}
Hz)')
        ax.patch.set_facecolor('#eefeee')

    ax.set_xlabel('Time (s)')
    ax.set_ylabel('Amplitude')
    ax.legend()
    ax.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

def wavelet_analysis(self, signal, signal_name="Signal"):
    """Perform wavelet analysis for time-frequency analysis"""

    # Continuous Wavelet Transform
    widths = np.arange(1, 50)

```

```

cwt_matrix = signal.cwt(signal, signal.morlet2, widths)

# Plot results
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(12, 8))

# Original signal
ax1.plot(signal)
ax1.set_title(f'{signal_name} - Time Domain')
ax1.set_xlabel('Time (hours)')
ax1.set_ylabel('Amplitude')

# Scalogram
im = ax2.imshow(np.abs(cwt_matrix), extent=[0, len(signal), 1,
50],
                 cmap='jet', aspect='auto')
ax2.set_title('Wavelet Transform - Time-Frequency Analysis')
ax2.set_xlabel('Time (hours)')
ax2.set_ylabel('Scale (related to frequency)')

plt.colorbar(im, ax=ax2, label='Magnitude')
plt.tight_layout()
plt.show()

# Practical application: Threat hunting with FFT
class ThreatHunterFFT:
    """Use FFT for threat hunting in network traffic"""

    def __init__(self):
        self.analyzer = SecuritySignalAnalyzer()

    def hunt_beaconing(self, traffic_data, min_beacon_period=0.5,
max_beacon_period=24):
        """Hunt for beaconing behavior in network traffic"""

        # Perform FFT
        N = len(traffic_data)
        fft_result = fft(traffic_data)
        frequencies = fftfreq(N, 1) # Assuming 1-hour sampling

        # Focus on positive frequencies
        positive_freq_idx = frequencies > 0
        frequencies = frequencies[positive_freq_idx]
        magnitude = np.abs(fft_result[positive_freq_idx])

        # Convert to periods
        periods = 1 / frequencies

        # Find peaks in the interesting period range
        period_mask = (periods >= min_beacon_period) & (periods <=
max_beacon_period)

```

```

relevant_magnitudes = magnitude[period_mask]
relevant_periods = periods[period_mask]

# Detect significant peaks
if len(relevant_magnitudes) > 0:
    threshold = np.mean(magnitude) + 3 * np.std(magnitude)
    peaks = relevant_magnitudes > threshold

beacons = []
for idx, is_peak in enumerate(peaks):
    if is_peak:
        beacon_period = relevant_periods[idx]
        beacon_strength = relevant_magnitudes[idx]

        # Calculate confidence based on signal strength
        confidence = min(beacon_strength / threshold, 1.0)

        beacons.append({
            'period_hours': beacon_period,
            'frequency': 1/beacon_period,
            'strength': beacon_strength,
            'confidence': confidence
        })

return beacons

```

116

```

def analyze_traffic_pattern(self, timestamps, values):
    """Analyze traffic patterns from timestamp-value pairs"""

    # Convert timestamps to hourly bins
    start_time = timestamps.min()
    hours_elapsed = (timestamps - start_time).dt.total_seconds() /
3600

    # Create hourly bins
    max_hours = int(hours_elapsed.max()) + 1
    hourly_counts = np.zeros(max_hours)

    for hour, value in zip(hours_elapsed.astype(int), values):
        hourly_counts[hour] += value

    # Analyze pattern
    beacons = self.hunt_beaconing(hourly_counts)

    return hourly_counts, beacons

# Demonstrate signal analysis
print("== Signal Analysis for Security ==\n")

```

```

# Create analyzer
analyzer = SecuritySignalAnalyzer()

# Generate and analyze signals
t, normal, beacon, ddos, total = analyzer.generate_security_signals()

# Analyze different components
print("== Analyzing Normal Traffic Pattern ==")
freq_normal, psd_normal, peaks_normal, periods_normal =
analyzer.analyze_frequency_spectrum(
    normal, "Normal Traffic Pattern"
)

print("\n== Analyzing Combined Signal (with hidden beaconing) ==")
freq_total, psd_total, peaks_total, periods_total =
analyzer.analyze_frequency_spectrum(
    total, "Combined Traffic (Normal + Beacon + DDoS)"
)

print("\n== Detected Periodic Components ==")
for period in periods_total[periods_total < 200]: # Ignore very long
    periods
    print(f"Period: {period:.1f} hours (Frequency: {1/period:.4f} cycles/hour)")

# Demonstrate aliasing
print("\n== Demonstrating Sampling and Aliasing ==")
analyzer.demonstrate_aliasing()

# Anomaly detection
print("\n== Sliding Window Anomaly Detection ==")
anomalies, scores, windows = analyzer.detect_periodic_anomalies(total)

# Plot anomaly detection results
plt.figure(figsize=(12, 6))
plt.subplot(2, 1, 1)
plt.plot(t, total)
for anomaly in anomalies:
    plt.axvspan(anomaly['start'], anomaly['end'], alpha=0.3, color='red')
plt.xlabel('Time (hours)')
plt.ylabel('Traffic')
plt.title('Traffic with Detected Anomalies')

plt.subplot(2, 1, 2)
window_times = [(w[0] + w[1])/2 for w in windows]
plt.plot(window_times, scores)
plt.axhline(np.mean(scores) + 2*np.std(scores), color='red', linestyle='--',
           label='Threshold')
plt.xlabel('Window Center (hours)')

```

```

plt.ylabel('Anomaly Score')
plt.title('Anomaly Scores Over Time')
plt.legend()

plt.tight_layout()
plt.show()

print(f"\nDetected {len(anomalies)} anomalous periods")
for i, anomaly in enumerate(anomalies):
    print(f"Anomaly {i+1}: Hours {anomaly['start']}-{anomaly['end']},"
Score: {anomaly['score']:.2f}")

# Threat hunting example
print("\n==== Threat Hunting with FFT ===")
hunter = ThreatHunterFFT()

# Create sample data with hidden beacon
np.random.seed(42)
hours = 168
base_traffic = np.random.poisson(100, hours)
beacon_traffic = 50 * np.sin(2 * np.pi * np.arange(hours) / 6) # 6-hour
beacon
beacon_traffic[beacon_traffic < 0] = 0
malicious_traffic = base_traffic + beacon_traffic

# Hunt for beacons
beacons_found = hunter.hunt_beaconing(malicious_traffic)

print("\n==== Beaconing Detection Results ===")
for beacon in beacons_found:
    print(f"Potential beacon detected:")
    print(f"  Period: {beacon['period_hours']:.2f} hours")
    print(f"  Confidence: {beacon['confidence']:.2%}")
    print(f"  Strength: {beacon['strength']:.2f}")

```

# Fundamental Concepts

## What is Hypothesis Testing?

A statistical method to make inferences about population parameters based on sample data.

## Key Components

### Null Hypothesis ( $H_0$ )

- Default assumption
- What we test against
- Example: "There is no increase in attack frequency"

### Alternative Hypothesis ( $H_1$ )

- What we want to prove
- Contradicts  $H_0$
- Example: "Attack frequency has increased"

## Test Statistic

- Calculated from sample data
- Follows known distribution under  $H_0$
- Used to make decision

119

## P-value

- Probability of observing test statistic (or more extreme) if  $H_0$  is true
- Small p-value → evidence against  $H_0$

## Significance Level ( $\alpha$ )

- Threshold for decision (commonly 0.05 or 0.01)
- If p-value <  $\alpha$ , reject  $H_0$

## Types of Errors

| Error Type | Description                | Consequence    | Example              |
|------------|----------------------------|----------------|----------------------|
| Type I     | Reject true $H_0$          | False positive | Alert when no attack |
| Type II    | Fail to reject false $H_0$ | False negative | Miss real attack     |

Reality vs Decision:

|              | $H_0$ True | $H_0$ False |
|--------------|------------|-------------|
| Reject $H_0$ | Type I     | ✓           |
| Don't Reject | ✓          | Type II     |

## Parametric Tests (assume distribution)

| Test                | Use Case            | Assumptions            | Example                 |
|---------------------|---------------------|------------------------|-------------------------|
| t-test              | Compare means       | Normal distribution    | Response time change    |
| ANOVA               | Compare 3+ groups   | Normal, equal variance | Multi-server comparison |
| Pearson correlation | Linear relationship | Normal, linear         | Traffic vs time         |

## Non-Parametric Tests (distribution-free)

| Test                 | Use Case               | Parametric Equivalent | Example               |
|----------------------|------------------------|-----------------------|-----------------------|
| Mann-Whitney U       | Compare medians        | t-test                | Skewed response times |
| Kruskal-Wallis       | Compare 3+ groups      | ANOVA                 | Non-normal groups     |
| Spearman correlation | Monotonic relationship | Pearson               | Rank-based analysis   |

121

## Effect Size

### Why Effect Size Matters

- P-value tells if effect exists
- Effect size tells if effect matters

### Common Effect Size Measures

| Measure    | Formula                                    | Interpretation                        |
|------------|--|---------------------------------------|
| Cohen's d  | $(\mu_1 - \mu_2) / \sigma_{\text{pooled}}$ | 0.2=small, 0.5=medium, 0.8=large      |
| $r^2$      | Explained variance                         | 0.01=small, 0.09=medium, 0.25=large   |
| Odds Ratio | $(a/b) / (c/d)$                            | 1=no effect, >1=positive, <1=negative |

## Multiple Testing Correction

When performing multiple tests, adjust for increased Type I error risk:

**Bonferroni Correction:**  $\alpha_{\text{adjusted}} = \alpha / n_{\text{tests}}$

**False Discovery Rate (FDR):** Controls expected proportion of false positives

```

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy import stats
import seaborn as sns
from statsmodels.stats.multicomp import pairwise_tukeyhsd
from statsmodels.stats.power import TTestPower

class StatisticalSecurityTester:
    """Statistical hypothesis testing for security metrics"""

    def __init__(self, significance_level=0.05):
        self.alpha = significance_level
        self.test_results = {}

    def normality_tests(self, data, data_name="Data"):
        """Test if data follows normal distribution"""

        tests = {
            'Shapiro-Wilk': stats.shapiro(data),
            'Anderson-Darling': stats.anderson(data),
            'Kolmogorov-Smirnov': stats.kstest(data, 'norm',
                                                args=(np.mean(data),
                                                   np.std(data)))
        }

        print(f"\n==== Normality Tests for {data_name} ====")

        # Shapiro-Wilk
        stat, p_value = tests['Shapiro-Wilk']
        print(f"Shapiro-Wilk Test:")
        print(f" Statistic: {stat:.4f}")
        print(f" P-value: {p_value:.4f}")
        print(f" Normal? {p_value > self.alpha}")

        # Anderson-Darling
        result = tests['Anderson-Darling']
        print(f"\nAnderson-Darling Test:")
        print(f" Statistic: {result.statistic:.4f}")
        for i, critical_value in enumerate(result.critical_values):
            sl = result.significance_level[i]
            print(f" Critical value at {sl}%%: {critical_value:.4f}")

        # Visual check
        fig, axes = plt.subplots(1, 3, figsize=(15, 5))

        # Histogram
        axes[0].hist(data, bins=30, density=True, alpha=0.7,

```

```

edgecolor='black')
x = np.linspace(data.min(), data.max(), 100)
axes[0].plot(x, stats.norm.pdf(x, np.mean(data), np.std(data)),
'r-', linewidth=2)
axes[0].set_title('Histogram with Normal Curve')

# Q-Q plot
stats.probplot(data, dist="norm", plot=axes[1])
axes[1].set_title('Q-Q Plot')

# Box plot
axes[2].boxplot(data)
axes[2].set_title('Box Plot')

plt.suptitle(f'Normality Assessment for {data_name}')
plt.tight_layout()
plt.show()

return p_value > self.alpha

def compare_two_groups(self, group1, group2, paired=False, names=
("Group1", "Group2")):
    """Compare two groups using appropriate test"""

print(f"\n==== Comparing {names[0]} vs {names[1]} ===")

# Check normality
normal1 = stats.shapiro(group1)[1] > self.alpha
normal2 = stats.shapiro(group2)[1] > self.alpha

if normal1 and normal2:
    # Use parametric test
    if paired:
        stat, p_value = stats.ttest_rel(group1, group2)
        test_name = "Paired t-test"
    else:
        # Check equal variances
        _, p_var = stats.levene(group1, group2)
        equal_var = p_var > self.alpha
        stat, p_value = stats.ttest_ind(group1, group2,
equal_var=equal_var)
        test_name = f"Independent t-test (equal_var={equal_var})"
    else:
        # Use non-parametric test
        if paired:
            stat, p_value = stats.wilcoxon(group1, group2)
            test_name = "Wilcoxon signed-rank test"
        else:
            stat, p_value = stats.mannwhitneyu(group1, group2,
alternative='two-sided')

```

```

        test_name = "Mann-Whitney U test"

        # Calculate effect size
        mean1, mean2 = np.mean(group1), np.mean(group2)
        pooled_std = np.sqrt((np.var(group1) + np.var(group2)) / 2)
        cohens_d = (mean1 - mean2) / pooled_std

        # Store results
        results = {
            'test': test_name,
            'statistic': stat,
            'p_value': p_value,
            'significant': p_value < self.alpha,
            'effect_size': cohens_d,
            'mean_difference': mean1 - mean2
        }

        print(f"Test used: {test_name}")
        print(f"Test statistic: {stat:.4f}")
        print(f"P-value: {p_value:.4f}")
        print(f"Significant? {results['significant']}")
        print(f"Effect size (Cohen's d): {cohens_d:.3f}")
        print(f"Mean difference: {mean1 - mean2:.3f}")

        # Visualize
        fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

        # Box plots
        ax1.boxplot([group1, group2], labels=names)
        ax1.set_ylabel('Value')
        ax1.set_title('Group Comparison')
        if results['significant']:
            ax1.text(1.5, ax1.get_ylim()[1] * 0.9, '***', ha='center',
            fontsize=20)

        # Violin plots with individual points
        data_combined = pd.DataFrame({
            'Value': np.concatenate([group1, group2]),
            'Group': [names[0]]*len(group1) + [names[1]]*len(group2)
        })
        sns.violinplot(data=data_combined, x='Group', y='Value', ax=ax2)
        sns.swarmplot(data=data_combined, x='Group', y='Value', ax=ax2,
                      color='black', alpha=0.5, size=3)
        ax2.set_title(f'{test_name}\n{p_value:.4f}')

        plt.tight_layout()
        plt.show()

    return results

```

```

def compare_multiple_groups(self, groups, group_names):
    """Compare multiple groups using ANOVA or Kruskal-Wallis"""

    print(f"\n==== Comparing {len(groups)} Groups ===")

    # Check normality for each group
    normality_results = []
    for i, group in enumerate(groups):
        _, p_norm = stats.shapiro(group)
        normality_results.append(p_norm > self.alpha)
        print(f"{group_names[i]} normal? {p_norm > self.alpha} (p={p_norm:.4f})")

    # Check homogeneity of variances
    _, p_var = stats.levene(*groups)
    equal_variances = p_var > self.alpha
    print(f"Equal variances? {equal_variances} (p={p_var:.4f})")

    if all(normality_results) and equal_variances:
        # Use one-way ANOVA
        stat, p_value = stats.f_oneway(*groups)
        test_name = "One-way ANOVA"

        # Post-hoc test if significant
        if p_value < self.alpha:
            # Prepare data for Tukey HSD
            data_stacked = []
            group_labels = []
            for i, group in enumerate(groups):
                data_stacked.extend(group)
                group_labels.extend([group_names[i]] * len(group))

            tukey_result = pairwise_tukeyhsd(data_stacked,
group_labels, alpha=self.alpha)
            print("\nPost-hoc Tukey HSD results:")
            print(tukey_result)
        else:
            # Use Kruskal-Wallis test
            stat, p_value = stats.kruskal(*groups)
            test_name = "Kruskal-Wallis test"

            # Post-hoc test if significant
            if p_value < self.alpha:
                print("\nPost-hoc pairwise comparisons (Mann-Whitney U):")
                from itertools import combinations
                for (i, name1), (j, name2) in
combinations(enumerate(group_names), 2):
                    _, p_pair = stats.mannwhitneyu(groups[i], groups[j])
                    # Bonferroni correction
                    p_adjusted = p_pair *

```

```

len(list(combinations(group_names, 2)))
    print(f" {name1} vs {name2}: p={p_pair:.4f} "
          f"(adjusted: {p_adjusted:.4f})")

print(f"\n{test_name} Results:")
print(f"Statistic: {stat:.4f}")
print(f"P-value: {p_value:.4f}")
print(f"Significant? {p_value < self.alpha}")

# Visualize
plt.figure(figsize=(10, 6))

# Create DataFrame for easier plotting
data_list = []
for i, group in enumerate(groups):
    for value in group:
        data_list.append({'Group': group_names[i], 'Value': value})
df = pd.DataFrame(data_list)

# Box plot with points
sns.boxplot(data=df, x='Group', y='Value', alpha=0.7)
sns.swarmplot(data=df, x='Group', y='Value', color='black',
alpha=0.5, size=3)

plt.title(f'{test_name}: p-value = {p_value:.4f}')
if p_value < self.alpha:
    plt.text(0.5, plt.ylim()[1] * 0.95, 'Significant difference
detected',
             ha='center', transform=plt.gca().transAxes,
             bbox=dict(boxstyle='round', facecolor='yellow',
alpha=0.5))
    plt.xticks(rotation=45)
    plt.tight_layout()
    plt.show()

return {'test': test_name, 'statistic': stat, 'p_value': p_value}

def time_series_change_detection(self, time_series,
change_point=None):
    """Detect if there's a significant change in time series"""

    if change_point is None:
        change_point = len(time_series) // 2

    before = time_series[:change_point]
    after = time_series[change_point:]

    print(f"\n==== Time Series Change Detection ===")
    print(f"Change point: {change_point}")

```

```

        print(f"Before: {len(before)} samples, After: {len(after)} samples")

        # Compare distributions
        results = self.compare_two_groups(before, after, paired=False,
                                           names= ("Before", "After"))

        # Additional tests
        # 1. Variance change (F-test)
        f_stat, p_var = stats.levene(before, after)
        print(f"\nVariance change test:")
        print(f"  F-statistic: {f_stat:.4f}")
        print(f"  P-value: {p_var:.4f}")
        print(f"  Significant variance change? {p_var < self.alpha}")

        # 2. Distribution change (KS test)
        ks_stat, p_ks = stats.ks_2samp(before, after)
        print(f"\nDistribution change test (KS):")
        print(f"  KS statistic: {ks_stat:.4f}")
        print(f"  P-value: {p_ks:.4f}")
        print(f"  Significant distribution change? {p_ks < self.alpha}")

        # Visualize
        fig, axes = plt.subplots(2, 2, figsize=(12, 10))

        # Time series plot
        ax1 = axes[0, 0]
        ax1.plot(time_series)
        ax1.axvline(x=change_point, color='red', linestyle='--',
label='Change point')
        ax1.set_xlabel('Time')
        ax1.set_ylabel('Value')
        ax1.set_title('Time Series with Change Point')
        ax1.legend()

        # Before/After histograms
        ax2 = axes[0, 1]
        ax2.hist(before, bins=30, alpha=0.5, label='Before', density=True)
        ax2.hist(after, bins=30, alpha=0.5, label='After', density=True)
        ax2.set_xlabel('Value')
        ax2.set_ylabel('Density')
        ax2.set_title('Distribution Comparison')
        ax2.legend()

        # Cumulative distributions
        ax3 = axes[1, 0]
        ax3.plot(np.sort(before), np.linspace(0, 1, len(before)),
label='Before')
        ax3.plot(np.sort(after), np.linspace(0, 1, len(after)),
label='After')

```

```

    ax3.set_xlabel('Value')
    ax3.set_ylabel('Cumulative Probability')
    ax3.set_title('Empirical CDF Comparison')
    ax3.legend()

    # Statistical summary
    ax4 = axes[1, 1]
    ax4.axis('off')
    summary = f"""
Change Detection Summary:

Before: μ={np.mean(before):.2f}, σ={np.std(before):.2f}
After:  μ={np.mean(after):.2f}, σ={np.std(after):.2f}

Mean change: {results['significant']} (p={results['p_value']:.4f})
Variance change: {p_var < self.alpha} (p={p_var:.4f})
Distribution change: {p_ks < self.alpha} (p={p_ks:.4f})

Effect size: {results['effect_size']:.3f}
"""

    ax4.text(0.1, 0.5, summary, transform=ax4.transAxes,
             fontsize=12, verticalalignment='center',
             fontfamily='monospace')

    plt.tight_layout()
    plt.show()

    return results

def power_analysis(self, effect_size=0.5, alpha=0.05, power=0.8):
    """Calculate required sample size for desired power"""

    analysis = TTestPower()

    # Calculate sample size needed
    n = analysis.solve_power(effect_size=effect_size, power=power,
                             alpha=alpha, alternative='two-sided')

    # Plot power curve
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

    # Power vs sample size
    sample_sizes = np.arange(10, 200, 5)
    powers = [analysis.solve_power(effect_size=effect_size, nobs=n,
                                   alpha=alpha, alternative='two-sided')
              for n in sample_sizes]

    ax1.plot(sample_sizes, powers, linewidth=2)
    ax1.axhline(y=0.8, color='red', linestyle='--', label='Target
power (0.8)')

```

```

        ax1.axvline(x=n, color='green', linestyle='--',
                     label=f'Required n={n:.0f}')
    ax1.set_xlabel('Sample Size')
    ax1.set_ylabel('Statistical Power')
    ax1.set_title(f'Power vs Sample Size (effect size={effect_size})')
    ax1.legend()
    ax1.grid(True, alpha=0.3)

    # Power vs effect size
    effect_sizes = np.linspace(0.1, 2.0, 50)
    powers_by_effect = [analysis.solve_power(effect_size=es, nobs=50,
                                              alpha=alpha,
                                              alternative='two-sided')
                         for es in effect_sizes]

    ax2.plot(effect_sizes, powers_by_effect, linewidth=2)
    ax2.axhline(y=0.8, color='red', linestyle='--', label='Target
power (0.8)')
    ax2.axvline(x=effect_size, color='green', linestyle='--',
                 label=f'Current effect size={effect_size}')
    ax2.set_xlabel('Effect Size (Cohen\'s d)')
    ax2.set_ylabel('Statistical Power')
    ax2.set_title('Power vs Effect Size (n=50)')
    ax2.legend()
    ax2.grid(True, alpha=0.3)

    plt.tight_layout()
    plt.show()

    print(f"\n==== Power Analysis Results ===")
    print(f"Effect size: {effect_size}")
    print(f"Significance level: {alpha}")
    print(f"Desired power: {power}")
    print(f"Required sample size: {n:.0f} per group")

    return n

# Demonstrate statistical testing
print("==== Statistical Hypothesis Testing for Security ===\n")

# Create test data
np.random.seed(42)

# Scenario 1: DDoS detection - comparing normal vs attack traffic
normal_traffic = np.random.normal(1000, 200, 100) # requests/second
ddos_traffic = np.random.normal(5000, 1000, 100) # much higher

# Scenario 2: Response time comparison between servers
server_a = np.random.lognormal(3, 0.5, 150) # milliseconds
server_b = np.random.lognormal(3.2, 0.6, 150)

```

```

server_c = np.random.lognormal(2.8, 0.4, 150)

# Scenario 3: Time series with change
baseline = np.random.normal(100, 20, 200)
attack_period = np.random.normal(150, 40, 100)
time_series = np.concatenate([baseline, attack_period])

# Initialize tester
tester = StatisticalSecurityTester()

# Test 1: Normal vs DDoS traffic
print("==> Test 1: DDoS Detection ==>")
ddos_results = tester.compare_two_groups(
    normal_traffic, ddos_traffic,
    paired=False,
    names=("Normal Traffic", "DDoS Traffic")
)

# Test 2: Multiple server comparison
print("\n==> Test 2: Server Performance Comparison ==>")
server_results = tester.compare_multiple_groups(
    [server_a, server_b, server_c],
    ["Server A", "Server B", "Server C"]
)

# Test 3: Change detection
print("\n==> Test 3: Time Series Change Detection ==>")
change_results = tester.time_series_change_detection(time_series,
change_point=200)

# Test 4: Power analysis
print("\n==> Test 4: Sample Size Planning ==>")
required_n = tester.power_analysis(effect_size=0.8, alpha=0.05, power=0.8)

# Summary report
print("\n==> Statistical Testing Summary Report ==>")
print(f"DDoS Detection: Traffic significantly different? {ddos_results['significant']}")  

print(f"  Effect size: {ddos_results['effect_size']:.2f} (very large)")  

print(f"Server Comparison: Significant differences? {server_results['p_value'] < 0.05}")  

print(f"Change Detection: Significant change detected? {change_results['significant']}")  

print(f"  Change magnitude: {change_results['mean_difference']:.1f} units")

```

## **Exercise 1: Statistical Analysis Pipeline**

Create a comprehensive analysis system that:

1. Loads security event data from multiple sources
2. Performs exploratory data analysis with all measures
3. Tests for distribution types and outliers
4. Compares metrics across different time periods
5. Generates automated reports with visualizations

## **Exercise 2: Bayesian Security System**

Build a Bayesian inference system that:

1. Starts with prior beliefs about threat levels
2. Updates beliefs as new evidence arrives
3. Handles multiple types of indicators
4. Provides uncertainty estimates
5. Visualizes belief evolution over time

## **Exercise 3: Advanced Signal Analysis**

132

Implement a signal processing toolkit that:

1. Detects multiple periodic patterns in network data
2. Identifies anomalous frequency components
3. Performs real-time FFT on streaming data
4. Correlates patterns across different signals
5. Generates alerts for suspicious periodicities

## **Exercise 4: Statistical Anomaly Detector**

Create an anomaly detection system that:

1. Learns baseline statistical properties
2. Uses multiple statistical tests for detection
3. Adjusts for multiple testing problems
4. Calculates confidence intervals for anomalies
5. Adapts to changing baselines

This section covered essential statistical and mathematical foundations:

## Key Concepts Mastered:

- **Descriptive Statistics:** Mean, median, mode, variance, and robust measures
- **Probability Theory:** Joint, conditional, and Bayes' theorem
- **Distributions:** Understanding and applying various probability distributions
- **Signal Analysis:** FFT for pattern detection and anomaly identification
- **Hypothesis Testing:** Making data-driven decisions with statistical rigor

## Important Formulas:

1. **Bayes' Theorem:**  $P(A|B) = P(B|A) \times P(A) / P(B)$
2. **Standard Deviation:**  $\sigma = \sqrt{(\sum(x_i - \mu)^2 / N)}$
3. **FFT Frequency:**  $f = k \times f_s / N$  ( $k=\text{bin}$ ,  $f_s=\text{sampling rate}$ ,  $N=\text{samples}$ )
4. **Cohen's d:**  $d = (\mu_1 - \mu_2) / \sigma_{\text{pooled}}$
5. **IQR Method:** Outlier if  $x < Q1 - 1.5 \times IQR$  or  $x > Q3 + 1.5 \times IQR$

## Critical Insights:

- Always check assumptions before applying statistical tests
- Use robust measures when dealing with security data (often has outliers)
- Periodic patterns in network traffic often indicate automated behavior
- Effect size matters as much as statistical significance
- Multiple testing requires correction to control false positives

133

## Next Steps:

In Section 3, we'll apply these statistical foundations to build machine learning models, starting with clustering algorithms and tree-based methods for classification and anomaly detection.

# Machine Learning Training Documentation

## Section 3: Machine Learning Essentials - Trees, Forests, & K-Means

### Overview

This section introduces fundamental machine learning algorithms, covering both unsupervised learning (clustering) and supervised learning (classification). We'll explore how these algorithms work, when to use them, and how to apply them to security analytics and threat detection.

### Learning Objectives

By the end of this section, you will be able to:

- Understand and implement clustering algorithms (K-Means, DBSCAN, Hierarchical)
- Apply dimensionality reduction techniques (PCA) effectively
- Build and optimize Support Vector Machines with appropriate kernels
- Create decision trees and understand their splitting criteria
- Implement Random Forests and other ensemble methods
- Choose the right algorithm for specific security use cases

# What is Clustering?

**Definition:** Clustering is an unsupervised learning technique that groups similar data points together without pre-labeled categories.

**Goal:** Find natural groupings where:

- **Intra-cluster similarity** is maximized (points within a cluster are similar)
- **Inter-cluster similarity** is minimized (clusters are distinct from each other)

## Types of Clustering Algorithms

Clustering Algorithm Taxonomy:

Partitioning Methods

- |— K-Means: Fixed K centroids
- |— K-Medoids: Actual points as centers
- |— K-Modes: For categorical data

Hierarchical Methods

- |— Agglomerative: Bottom-up (merge)
- |— Divisive: Top-down (split)
- |— Produces: Dendrogram

135

Density-Based Methods

- |— DBSCAN: Density-connected regions
- |— OPTICS: Ordering points
- |— HDBSCAN: Hierarchical DBSCAN

Distribution-Based Methods

- |— Gaussian Mixture Models
- |— Expectation Maximization

## Clustering Algorithm Comparison

| Algorithm | Cluster Shape | Outliers  | Need K? | Scalability | Use Case                 |
|-----------|---------------|-----------|---------|-------------|--------------------------|
| K-Means   | Spherical     | Sensitive | Yes     | Excellent   | Equal-sized clusters     |
| K-Medoids | Spherical     | Robust    | Yes     | Poor        | Small data with outliers |
| DBSCAN    | Arbitrary     | Detects   | No      | Good        | Varying densities        |

| Algorithm    | Cluster Shape | Outliers  | Need K? | Scalability | Use Case             |
|--------------|---------------|-----------|---------|-------------|----------------------|
| Hierarchical | Arbitrary     | Sensitive | No*     | Poor        | Taxonomy needed      |
| GMM          | Elliptical    | Handles   | Yes     | Good        | Overlapping clusters |

\*Can cut dendrogram at desired level

## Distance Metrics

**Key Concept:** Clustering quality depends heavily on the distance metric used.

| Metric    | Formula                           | Use When               | Sensitive To |
|-----------|-----------------------------------|------------------------|--------------|
| Euclidean | $\sqrt{\sum(x_i - y_i)^2}$        | Continuous features    | Scale        |
| Manhattan | $\sum x_i - y_i $                 | Grid-like data         | Scale        |
| Cosine    | $1 - (X \cdot Y) / (\ X\  \ Y\ )$ | Text, high-dimensional | Magnitude    |
| Hamming   | Count( $x_i \neq y_i$ )           | Binary/categorical     | N/A          |

# Algorithm Overview

**K-Means** partitions n observations into K clusters by minimizing within-cluster variance.

**Objective Function (Sum of Squared Errors):**

$$J = \sum_{i=1}^n \sum_{j=1}^k w_{ij} \|x_i - \mu_j\|^2$$

Where:

- $w_{ij} = 1$  if  $x_i$  belongs to cluster j, 0 otherwise
- $\mu_j$  = centroid of cluster j

## K-Means Algorithm Steps

1. Initialize: Choose K random centroids
2. Assign: Each point to nearest centroid
3. Update: Recalculate centroids as mean of assigned points
4. Repeat: Until convergence (centroids don't move)

## K-Means Characteristics

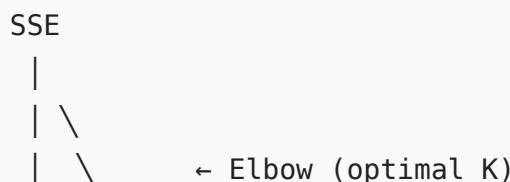
137

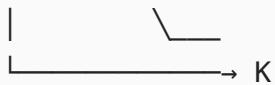
| Characteristic                     | Description                       | Implication                |
|------------------------------------|-----------------------------------|----------------------------|
| <b>Assumes spherical clusters</b>  | Uses Euclidean distance           | Poor on elongated clusters |
| <b>Sensitive to initialization</b> | Random start affects result       | Use multiple runs          |
| <b>Sensitive to outliers</b>       | Mean affected by extremes         | Consider K-Medoids         |
| <b>Requires K upfront</b>          | Must specify clusters             | Use elbow method           |
| <b>Scales well</b>                 | $O(n \cdot k \cdot t)$ complexity | Good for large datasets    |

## Choosing Optimal K

### 1. Elbow Method

Plot SSE vs K and look for "elbow":





## 2. Silhouette Score

Measures how similar a point is to its cluster vs other clusters:

$$s(i) = (b(i) - a(i)) / \max(a(i), b(i))$$

Where:

- $a(i)$  = average distance to points in same cluster
- $b(i)$  = average distance to points in nearest cluster
- Range:  $[-1, 1]$  (higher is better)

## 3. Gap Statistic

Compares within-cluster dispersion to random data.

- "K-Means finds K center points by averaging"
- "Like finding the center of mass for K groups"
- "Voronoi diagram with K cells"

## K-Means Variants

| Variant                   | Modification         | Advantage            | Use Case       |
|---------------------------|----------------------|----------------------|----------------|
| <b>K-Means++</b>          | Smart initialization | Better convergence   | Default choice |
| <b>Mini-Batch K-Means</b> | Sample batches       | Faster on large data | Big data       |
| <b>K-Medoids</b>          | Use actual points    | Robust to outliers   | Noisy data     |
| <b>K-Modes</b>            | Mode for categorical | Handles categories   | Mixed data     |

---

# Algorithm Concept

KNN is a "lazy" algorithm that classifies points based on the class of their neighbors.

**Key Principle:** "You are who you hang out with"

## KNN for Classification

For a new point  $x$ :

1. Find K nearest neighbors
2. Count class frequencies
3. Assign majority class

**Decision Rule:**

$$\hat{y} = \underset{c}{\operatorname{argmax}} \sum_{i \in N_k(x)} I(y_i = c)$$

Where  $N_k(x) = K$  nearest neighbors of  $x$

## KNN Characteristics

140

| Aspect                | Description             | Impact                         |
|-----------------------|-------------------------|--------------------------------|
| <b>Non-parametric</b> | No model parameters     | Flexible boundaries            |
| <b>Lazy learning</b>  | No training phase       | Fast training, slow prediction |
| <b>Local method</b>   | Uses nearby points only | Captures local patterns        |
| <b>Memory-based</b>   | Stores all data         | High memory usage              |

## Choosing K

| K Value              | Effect                   | Use When         |
|----------------------|--------------------------|------------------|
| <b>K = 1</b>         | Highest variance         | Very low noise   |
| <b>K small (3-7)</b> | Flexible boundary        | Complex patterns |
| <b>K large</b>       | Smooth boundary          | High noise       |
| <b>K = n</b>         | Always predicts majority | Never!           |

**Rule of thumb:**  $K = \sqrt{n}$  (odd number to avoid ties)



# Core Concepts

**DBSCAN** groups points that are closely packed together, marking outliers as noise.

## Point Types in DBSCAN

| Core Point: | Border Point: | Noise Point: |
|-------------|---------------|--------------|
| ●           | ○             | ✗            |
| ● ● ●       | ● ○           |              |
| ● ● ● ●     | ● ● ●         | ✗            |
| ● ● ●       | ● ●           |              |

# DBSCAN Parameters

| Parameter | Symbol        | Meaning             | How to Choose          |
|-----------|---------------|---------------------|------------------------|
| Epsilon   | $\varepsilon$ | Neighborhood radius | k-distance plot        |
| MinPts    | MinPts        | Min points for core | 2×dimensions (minimum) |

# DBSCAN Algorithm

1. For each point  $p$ :
    - If  $p$  is core point  $\rightarrow$  start new cluster
    - If  $p$  is border point  $\rightarrow$  assign to cluster
    - If  $p$  is noise  $\rightarrow$  mark as outlier
  2. Expand clusters:
    - Add all density-reachable points
    - Include border points

# DBSCAN vs K-Means

| Feature                   | DBSCAN           | K-Means              |
|---------------------------|------------------|----------------------|
| <b>Cluster shape</b>      | Arbitrary        | Spherical only       |
| <b>Number of clusters</b> | Automatic        | Must specify K       |
| <b>Outlier handling</b>   | Identifies noise | Forces into clusters |

| Feature      | DBSCAN          | K-Means                |
|--------------|-----------------|------------------------|
| Cluster size | Varying         | Tends to equal         |
| Performance  | $O(n \log n)^*$ | $O(n \cdot k \cdot t)$ |

\*With spatial index

## When to Use DBSCAN

### Use when:

- Clusters have arbitrary shapes
- Data contains outliers
- Don't know number of clusters
- Clusters have varying densities

### Avoid when:

- High-dimensional data
- Clusters have very different densities
- Need exact K clusters

# What is PCA?

PCA is a dimensionality reduction technique that finds the directions of maximum variance in data.

**Goal:** Transform data to a new coordinate system where:

1. First axis explains most variance
2. Second axis explains second-most variance
3. All axes are orthogonal (uncorrelated)

## Mathematical Foundation

**PCA Steps:**

1. **Standardize** data:  $X' = (X - \mu) / \sigma$
2. **Compute** covariance matrix:  $C = (1/n)X'TX'$
3. **Find** eigenvectors and eigenvalues of C
4. **Sort** by eigenvalues (descending)
5. **Project** data onto top k eigenvectors

**Transform:**  $Y = XW$

Where W = matrix of selected eigenvectors

144

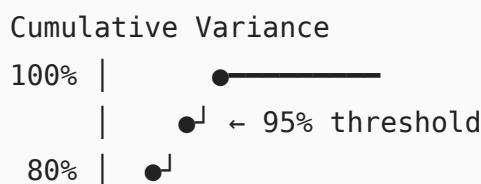
## PCA Properties

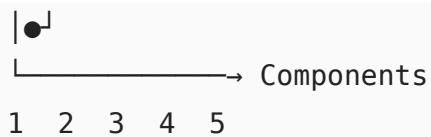
| Property                   | Description                    | Implication                      |
|----------------------------|--------------------------------|----------------------------------|
| <b>Linear</b>              | Linear combination of features | Can't capture nonlinear patterns |
| <b>Orthogonal</b>          | Components uncorrelated        | Removes redundancy               |
| <b>Variance-maximizing</b> | Orders by importance           | Can truncate dimensions          |
| <b>Unsupervised</b>        | No labels needed               | Finds structure blindly          |

## Choosing Number of Components

### 1. Explained Variance Ratio

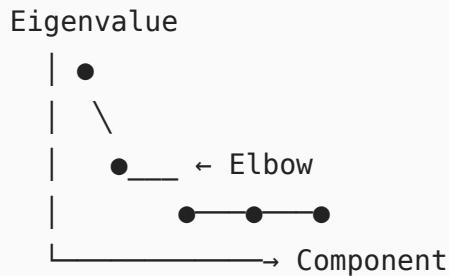
Keep k components that explain  $\geq 95\%$  variance:





## 2. Scree Plot

Look for "elbow" in eigenvalues:



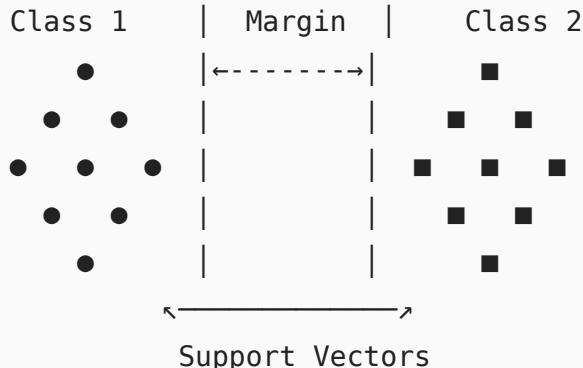
## PCA for Anomaly Detection

**Method:** Reconstruction error

1. Project to lower dimension:  $Y = XW$
2. Reconstruct:  $\hat{X} = YW^T$
3. Error:  $e = \|X - \hat{X}\|^2$
4. Anomaly if  $e > \text{threshold}$

# Core Concept

SVM finds the optimal hyperplane that maximizes the margin between classes.



## SVM Optimization Problem

Primal Form:

$$\begin{aligned} \text{minimize: } & (1/2) \|w\|^2 + C \sum \xi_i \\ \text{subject to: } & y_i (w^T x_i + b) \geq 1 - \xi_i \\ & \xi_i \geq 0 \end{aligned}$$

146

Where:

- $w$  = weight vector
- $b$  = bias
- $\xi_i$  = slack variables
- $C$  = regularization parameter

## Types of SVM

| Type          | When to Use         | Characteristics   |
|---------------|---------------------|-------------------|
| Linear SVM    | Linearly separable  | Fast, simple      |
| Kernel SVM    | Non-linear patterns | Flexible, slower  |
| One-Class SVM | Anomaly detection   | Unsupervised      |
| SVR           | Regression          | Continuous output |

## Kernel Functions

The Kernel Trick: Map data to higher dimension implicitly

| Kernel         | Formula                            | Use Case              | Parameters     |
|----------------|------------------------------------|-----------------------|----------------|
| Linear         | $K(x,y) = x^T y$                   | Linearly separable    | None           |
| Polynomial     | $K(x,y) = (yx^T y + r)^d$          | Polynomial boundaries | $d, \gamma, r$ |
| RBF (Gaussian) | $K(x,y) = \exp(-\gamma \ x-y\ ^2)$ | Most common, flexible | $\gamma$       |
| Sigmoid        | $K(x,y) = \tanh(\gamma x^T y + r)$ | Neural network-like   | $\gamma, r$    |

## SVM Hyperparameters

| Parameter        | Effect                              | How to Choose              |
|------------------|-------------------------------------|----------------------------|
| C                | Trade-off between margin and errors | Cross-validation           |
| $\gamma$ (gamma) | RBF kernel width                    | Start with $1/n\_features$ |
| degree           | Polynomial degree                   | Usually 2-5                |

## SVM Characteristics

### ✓ Advantages:

- Effective in high dimensions
- Memory efficient (only stores support vectors)
- Versatile with different kernels
- Strong theoretical foundation

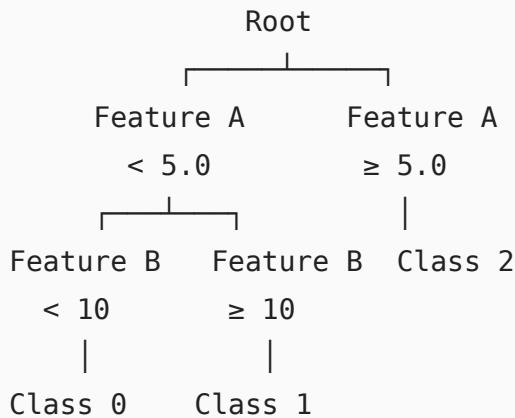
147

### ✗ Disadvantages:

- Slow on large datasets
- Sensitive to parameter tuning
- No probability estimates (by default)
- Doesn't handle imbalanced data well

# How Decision Trees Work

Decision Trees recursively split data based on feature values to create a tree-like model.



## Splitting Criteria

### 1. Gini Impurity (Classification)

$$\text{Gini}(S) = 1 - \sum_i p_i^2$$

Where  $p_i$  = proportion of class  $i$

148

### 2. Entropy (Classification)

$$\text{Entropy}(S) = -\sum_i p_i \log_2(p_i)$$

### 3. Information Gain

$$IG(S, A) = \text{Entropy}(S) - \sum_{v \in \text{Values}(A)} \left( \frac{|S_v|}{|S|} \right) \times \text{Entropy}(S_v)$$

### 4. MSE (Regression)

$$MSE = (1/n) \sum (y_i - \hat{y})^2$$

## Splitting Algorithm

Best split maximizes:

- Information Gain (ID3)
- Gain Ratio (C4.5)
- Gini decrease (CART)

| Parameter                | Purpose              | Effect of Increasing       |
|--------------------------|----------------------|----------------------------|
| <b>max_depth</b>         | Tree depth limit     | More complex, overfit risk |
| <b>min_samples_split</b> | Min samples to split | Simpler tree               |
| <b>min_samples_leaf</b>  | Min samples in leaf  | Simpler tree               |
| <b>max_features</b>      | Features to consider | More options, slower       |

## Advantages vs Disadvantages

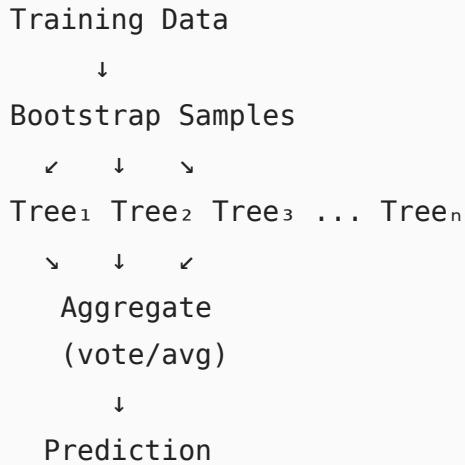
| Advantages         | Disadvantages                             |
|--------------------|---|
| Easy to interpret  | Prone to overfitting                      |
| Handles non-linear | Unstable (small changes → different tree) |
| No scaling needed  | Biased to features with more levels       |
| Feature importance | Poor on linear relationships              |

## Memory Trick

"Decision trees ask yes/no questions to reach a decision"

# Ensemble Learning Concept

**Random Forest** = Many decision trees + Bagging + Feature randomness



## Random Forest Algorithm

1. **Bootstrap:** Sample n examples with replacement
2. **Random features:** At each split, consider random subset of features
3. **Grow tree:** No pruning (grow deep)
4. **Repeat:** Build many trees (100-1000)
5. **Aggregate:**
  - Classification: Majority vote
  - Regression: Average

151

## Key Innovations

| Innovation                | Effect                       | Benefit            |
|---------------------------|------------------------------|--------------------|
| <b>Bagging</b>            | Different data per tree      | Reduces variance   |
| <b>Feature randomness</b> | Different features per split | Decorrelates trees |
| <b>No pruning</b>         | Deep trees                   | Low bias           |
| <b>Ensemble</b>           | Many trees                   | Stable predictions |

## Random Forest Parameters

| Parameter           | Default | Effect          | Tuning Tip                           |
|---------------------|---------|-----------------|--------------------------------------|
| <b>n_estimators</b> | 100     | Number of trees | More is better (diminishing returns) |

| Parameter                | Default              | Effect             | Tuning Tip             |
|--------------------------|----------------------|--------------------|------------------------|
| <b>max_features</b>      | $\sqrt{n\_features}$ | Features per split | Lower = more random    |
| <b>max_depth</b>         | None                 | Tree depth         | Control overfitting    |
| <b>min_samples_split</b> | 2                    | Min to split       | Higher = simpler trees |

**Unique to Random Forest:** Each tree has ~37% unseen data

For each sample:

1. Find trees that didn't use it
  2. Get their predictions
  3. Calculate error
- Free validation set!

## Feature Importance

Two methods:

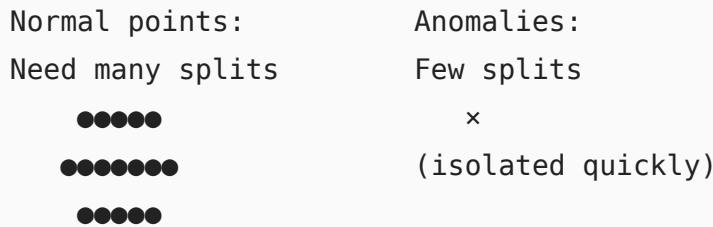
1. **Mean Decrease Impurity:** Average impurity decrease across all trees
2. **Permutation Importance:** Decrease in accuracy when feature is shuffled

## Random Forest vs Single Tree

| Aspect           | Decision Tree | Random Forest |
|------------------|---------------|---------------|
| Accuracy         | Lower         | Higher        |
| Overfitting      | High risk     | Low risk      |
| Interpretability | Easy          | Hard          |
| Training time    | Fast          | Slower        |
| Prediction time  | Fast          | Slower        |

# Isolation Forest

**Key Insight:** Anomalies are easier to isolate than normal points



**Anomaly Score:**

$$s(x) = 2^{-E(h(x))/c(n)}$$

Where:

- $E(h(x))$  = average path length
- $c(n)$  = average path length of BST

# One-Class SVM

**Concept:** Learn boundary around normal data

154

**Optimization:**

$$\begin{aligned} \min \quad & (1/2) \|w\|^2 - \rho + (1/vn)\sum \xi_i \\ \text{s.t.} \quad & (w \cdot \Phi(x_i)) \geq \rho - \xi_i \end{aligned}$$

Where  $v$  = fraction of outliers

| Strategy         | Method                    | Advantage      |
|------------------|---------------------------|----------------|
| <b>Voting</b>    | Combine binary decisions  | Simple, robust |
| <b>Averaging</b> | Average anomaly scores    | Smooth scores  |
| <b>Stacking</b>  | Learn combination weights | Optimal fusion |
| <b>Cascading</b> | Sequential filtering      | Efficient      |

## Comparison of Anomaly Detection Methods

| Method                  | Assumption           | Scalability | Interpretability |
|-------------------------|----------------------|-------------|------------------|
| <b>Statistical</b>      | Known distribution   | Excellent   | High             |
| <b>Isolation Forest</b> | Isolation property   | Good        | Medium           |
| <b>One-Class SVM</b>    | Margin exists        | Poor        | Low              |
| <b>Autoencoder</b>      | Reconstruction error | Good        | Low              |
| <b>DBSCAN</b>           | Density differences  | Medium      | High             |

# Algorithm Selection Guide

Start Here

↓

Is data labeled?

| └ No → Unsupervised

| | └ Finding groups? → Clustering

| | | └ Know K? → K-Means

| | | └ Arbitrary shapes? → DBSCAN

| | | └ Hierarchical? → Agglomerative

| | └ Reducing dimensions? → PCA

| └ Yes → Supervised

| └ Classification

| | └ Linear? → Logistic/Linear SVM

| | └ Non-linear? → Kernel SVM/Trees

| | | └ Need probabilities? → Random Forest

| └ Anomaly Detection → One-Class SVM/Isolation Forest

## Implementation Code Examples

Now let's see these algorithms in action with security-focused examples:

156

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.cluster import KMeans, DBSCAN, AgglomerativeClustering
from sklearn.neighbors import KNeighborsClassifier
from sklearn.decomposition import PCA
from sklearn.svm import SVC, OneClassSVM
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.ensemble import RandomForestClassifier, IsolationForest
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import silhouette_score, confusion_matrix,
classification_report
from sklearn.model_selection import train_test_split, GridSearchCV
import warnings
warnings.filterwarnings('ignore')

# Set random seed for reproducibility
np.random.seed(42)

class SecurityMLToolkit:
    """Comprehensive toolkit for security ML applications"""
```

```

def __init__(self):
    self.scaler = StandardScaler()
    self.models = {}
    self.results = {}

def generate_security_dataset(self, scenario='network'):
    """Generate synthetic security data for demonstrations"""

    if scenario == 'network':
        # Network traffic patterns
        n_normal = 1000
        n_suspicious = 300
        n_attack = 200

        # Normal traffic: clustered around typical values
        normal_traffic = np.random.multivariate_normal(
            mean=[100, 5000, 20, 0.1], # packet_rate, bytes,
duration, error_rate
            cov=[[400, 1000, 5, 0.01],
                  [1000, 1000000, 100, 0.05],
                  [5, 100, 25, 0.001],
                  [0.01, 0.05, 0.001, 0.01]],
            size=n_normal
        )

        # Suspicious: higher variance
        suspicious_traffic = np.random.multivariate_normal(
            mean=[300, 15000, 60, 0.3],
            cov=[[2500, 5000, 20, 0.1],
                  [5000, 9000000, 500, 0.2],
                  [20, 500, 400, 0.01],
                  [0.1, 0.2, 0.01, 0.04]],
            size=n_suspicious
        )

        # Attack: distinct patterns
        attack_traffic = np.concatenate([
            # DDoS pattern
            np.random.multivariate_normal(
                mean=[1000, 1000, 5, 0.8],
                cov=[[10000, 100, 1, 0.1],
                      [100, 10000, 10, 0.1],
                      [1, 10, 4, 0.01],
                      [0.1, 0.1, 0.01, 0.04]],
                size=n_attack//2
            ),
            # Data exfiltration pattern
            np.random.multivariate_normal(
                mean=[50, 100000, 300, 0.01],
                cov=[[100, 10000, 50, 0.001],

```

```

        [10000, 100000000, 1000, 0.01],
        [50, 1000, 2500, 0.001],
        [0.001, 0.01, 0.001, 0.0001]],
        size=n_attack//2
    )
])

# Combine data
X = np.vstack([normal_traffic, suspicious_traffic,
attack_traffic])
y = np.array(['normal']*n_normal + ['suspicious']*n_suspicious +
['attack']*n_attack)

# Ensure positive values
X = np.abs(X)

feature_names = ['packet_rate', 'bytes_transferred',
'duration_seconds', 'error_rate']

return X, y, feature_names

elif scenario == 'user_behavior':
    # User behavior patterns
    n_users = 500

    # Normal users
    normal_users = np.random.multivariate_normal(
        mean=[8, 50, 100, 20], # login_hour, files_accessed,
    keystrokes/min, apps_used
        cov=[[4, 5, 10, 2],
              [5, 225, 50, 10],
              [10, 50, 400, 20],
              [2, 10, 20, 25]],
        size=int(n_users * 0.9)
    )

    # Insider threats
    insider_threats = np.random.multivariate_normal(
        mean=[22, 200, 150, 5], # Late hours, many files, fast
    typing, few apps
        cov=[[9, 10, 20, 1],
              [10, 2500, 100, 5],
              [20, 100, 900, 10],
              [1, 5, 10, 4]],
        size=int(n_users * 0.1)
    )

    X = np.vstack([normal_users, insider_threats])
    y = np.array(['normal']*len(normal_users) +
['threat']*len(insider_threats))

```

```

# Wrap hours to 0-24
X[:, 0] = X[:, 0] % 24
X = np.abs(X)

feature_names = ['login_hour', 'files_accessed',
'keystrokes_per_min', 'apps_used']

return X, y, feature_names

def demonstrate_kmeans(self, X, feature_names):
    """Demonstrate K-Means clustering"""
    print("== K-Means Clustering Demo ==\n")

    # Scale data
    X_scaled = self.scaler.fit_transform(X)

    # Find optimal K using elbow method
    inertias = []
    silhouette_scores = []
    K_range = range(2, 10)

    for k in K_range:
        kmeans = KMeans(n_clusters=k, random_state=42, n_init=10)
        kmeans.fit(X_scaled)
        inertias.append(kmeans.inertia_)
        silhouette_scores.append(silhouette_score(X_scaled,
kmeans.labels_))

    # Visualize elbow method
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

    ax1.plot(K_range, inertias, 'bo-')
    ax1.set_xlabel('Number of Clusters (K)')
    ax1.set_ylabel('Inertia (Within-cluster sum of squares)')
    ax1.set_title('Elbow Method')
    ax1.grid(True, alpha=0.3)

    ax2.plot(K_range, silhouette_scores, 'ro-')
    ax2.set_xlabel('Number of Clusters (K)')
    ax2.set_ylabel('Silhouette Score')
    ax2.set_title('Silhouette Analysis')
    ax2.grid(True, alpha=0.3)

    plt.tight_layout()
    plt.show()

    # Use optimal K (let's say 3 for this example)
    optimal_k = 3
    kmeans = KMeans(n_clusters=optimal_k, random_state=42, n_init=10)

```

```

labels = kmeans.fit_predict(X_scaled)

# Analyze clusters
print(f"Optimal K: {optimal_k}")
print("\nCluster Characteristics:")
for i in range(optimal_k):
    cluster_data = X[labels == i]
    print(f"\nCluster {i} ({len(cluster_data)} samples):")
    for j, feature in enumerate(feature_names):
        print(f"  {feature}: mean={cluster_data[:, j].mean():.2f},"
std={cluster_data[:, j].std():.2f}")

# Visualize clusters (using first 2 principal components)
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_scaled)

plt.figure(figsize=(10, 8))
scatter = plt.scatter(X_pca[:, 0], X_pca[:, 1], c=labels,
cmap='viridis', alpha=0.6)
centers_pca = pca.transform(kmeans.cluster_centers_)
plt.scatter(centers_pca[:, 0], centers_pca[:, 1], c='red', s=200,
marker='*', edgecolor='black')
plt.xlabel(f'PC1 ({pca.explained_variance_ratio_[0]:.2%} variance)')
plt.ylabel(f'PC2 ({pca.explained_variance_ratio_[1]:.2%} variance)')
plt.title('K-Means Clustering Results')
plt.colorbar(scatter, label='Cluster')
plt.grid(True, alpha=0.3)
plt.show()

self.models['kmeans'] = kmeans
return labels

def demonstrate_dbSCAN(self, X, feature_names):
    """Demonstrate DBSCAN clustering"""
    print("\n==== DBSCAN Clustering Demo ===\n")

    # Scale data
    X_scaled = self.scaler.fit_transform(X)

    # Find optimal parameters using k-distance graph
    from sklearn.neighbors import NearestNeighbors

    # Calculate k-distances
    k = 5 # MinPts - 1
    nbrs = NearestNeighbors(n_neighbors=k).fit(X_scaled)
    distances, indices = nbrs.kneighbors(X_scaled)
    k_distances = distances[:, k-1]
    k_distances = np.sort(k_distances)

```

```

# Plot k-distance graph
plt.figure(figsize=(10, 6))
plt.plot(k_distances)
plt.xlabel('Points sorted by distance')
plt.ylabel(f'{k}-NN Distance')
plt.title('K-Distance Graph for Epsilon Selection')
plt.grid(True, alpha=0.3)

# Add elbow annotation
elbow_idx = len(k_distances) - 50 # Approximate elbow
plt.axhline(y=k_distances[elbow_idx], color='red', linestyle='--',
label=f'\u03b5 \u2248 {k_distances[elbow_idx]:.2f}')
plt.legend()
plt.show()

# Apply DBSCAN
eps = k_distances[elbow_idx]
dbscan = DBSCAN(eps=eps, min_samples=5)
labels = dbscan.fit_predict(X_scaled)

# Analyze results
n_clusters = len(set(labels)) - (1 if -1 in labels else 0)
n_noise = list(labels).count(-1)

print(f"Parameters: eps={eps:.3f}, min_samples=5")
print(f"Number of clusters: {n_clusters}")
print(f"Number of noise points: {n_noise}
({n_noise/len(X)*100:.1f}%)")

# Cluster characteristics
for i in range(n_clusters):
    cluster_data = X[labels == i]
    print(f"\nCluster {i} ({len(cluster_data)} samples):")
    for j, feature in enumerate(feature_names):
        print(f"  {feature}: mean={cluster_data[:, j].mean():.2f},
std={cluster_data[:, j].std():.2f}")

# Visualize
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_scaled)

plt.figure(figsize=(10, 8))
unique_labels = set(labels)
colors = plt.cm.viridis(np.linspace(0, 1, len(unique_labels)))

for k, col in zip(unique_labels, colors):
    if k == -1:
        col = 'red'
        marker = 'x'

```

```

        label = 'Noise'
    else:
        marker = 'o'
        label = f'Cluster {k}'

    class_member_mask = (labels == k)
    xy = X_pca[class_member_mask]
    plt.scatter(xy[:, 0], xy[:, 1], c=[col], marker=marker,
                s=50 if marker == 'o' else 100, label=label,
alpha=0.6)

    plt.xlabel(f'PC1 ({pca.explained_variance_ratio_[0]:.2%} variance)')
    plt.ylabel(f'PC2 ({pca.explained_variance_ratio_[1]:.2%} variance)')
    plt.title('DBSCAN Clustering Results')
    plt.legend()
    plt.grid(True, alpha=0.3)
    plt.show()

self.models['dbscan'] = dbscan
return labels

def demonstrate_pca(self, X, y, feature_names):
    """Demonstrate PCA for dimensionality reduction"""
    print("\n==== PCA Dimensionality Reduction Demo ====\n")

    # Scale data
    X_scaled = self.scaler.fit_transform(X)

    # Apply PCA
    pca = PCA()
    X_pca = pca.fit_transform(X_scaled)

    # Explained variance
    cumsum_var = np.cumsum(pca.explained_variance_ratio_)

    # Visualize explained variance
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

    # Scree plot
    ax1.bar(range(1, len(pca.explained_variance_ratio_) + 1),
            pca.explained_variance_ratio_)
    ax1.set_xlabel('Principal Component')
    ax1.set_ylabel('Explained Variance Ratio')
    ax1.set_title('Scree Plot')
    ax1.grid(True, alpha=0.3, axis='y')

    # Cumulative variance
    ax2.plot(range(1, len(cumsum_var) + 1), cumsum_var, 'bo-')

```

```

        ax2.axhline(y=0.95, color='red', linestyle='--', label='95%
variance')
        ax2.set_xlabel('Number of Components')
        ax2.set_ylabel('Cumulative Explained Variance')
        ax2.set_title('Cumulative Explained Variance')
        ax2.legend()
        ax2.grid(True, alpha=0.3)

    plt.tight_layout()
    plt.show()

# Find components for 95% variance
n_components_95 = np.argmax(cumsum_var >= 0.95) + 1
print(f"Components needed for 95% variance: {n_components_95} (out
of {len(feature_names)}")
print(f"Dimension reduction: {(1 -
n_components_95/len(feature_names))*100:.1f}%")

# Show component loadings
print("\nPrincipal Component Loadings:")
components_df = pd.DataFrame(
    pca.components_[:,2].T,
    columns=['PC1', 'PC2'],
    index=feature_names
)
print(components_df)

# Visualize in 2D
plt.figure(figsize=(10, 8))

# Map string labels to numeric
unique_labels = np.unique(y)
label_map = {label: i for i, label in enumerate(unique_labels)}
y_numeric = np.array([label_map[label] for label in y])

scatter = plt.scatter(X_pca[:, 0], X_pca[:, 1], c=y_numeric,
                      cmap='viridis', alpha=0.6)
plt.xlabel(f'PC1 ({pca.explained_variance_ratio_[0]:.2%}
variance)')
plt.ylabel(f'PC2 ({pca.explained_variance_ratio_[1]:.2%}
variance)')
plt.title('PCA: 2D Visualization of High-Dimensional Data')

# Create custom legend
handles = []
for label, idx in label_map.items():
    handles.append(plt.scatter([], [], c=plt.cm.viridis(idx/len(unique_labels)),
                           s=50, label=label))
plt.legend(handles=handles, title='Class')

```

```
plt.grid(True, alpha=0.3)
plt.show()

# Demonstrate anomaly detection with PCA
print("\n==== PCA for Anomaly Detection ===")

# Use 2 components for reconstruction
pca_2 = PCA(n_components=2)
X_reduced = pca_2.fit_transform(X_scaled)
X_reconstructed = pca_2.inverse_transform(X_reduced)

# Calculate reconstruction error
reconstruction_error = np.sum((X_scaled - X_reconstructed)**2,
axis=1)

# Find threshold (95th percentile)
threshold = np.percentile(reconstruction_error, 95)
anomalies = reconstruction_error > threshold

print(f"Anomalies detected: {np.sum(anomalies)}
({np.sum(anomalies)/len(X)*100:.1f}%)")

# Visualize anomalies
plt.figure(figsize=(10, 6))
plt.scatter(range(len(reconstruction_error)),
reconstruction_error,
c=['red' if a else 'blue' for a in anomalies],
alpha=0.5)
plt.axhline(y=threshold, color='red', linestyle='--',
label=f'Threshold (95th percentile)')
plt.xlabel('Sample Index')
plt.ylabel('Reconstruction Error')
plt.title('PCA Anomaly Detection')
plt.legend()
plt.grid(True, alpha=0.3)
plt.show()

self.models['pca'] = pca
return X_pca

def demonstrate_svm(self, X, y, feature_names):
    """Demonstrate SVM classification"""
    print("\n==== SVM Classification Demo ===\n")

    # For binary classification, use only two classes
    mask = (y == 'normal') | (y == 'attack')
    X_binary = X[mask]
    y_binary = y[mask]

    # Scale and split data
```

```

X_scaled = self.scaler.fit_transform(X_binary)
X_train, X_test, y_train, y_test = train_test_split(
    X_scaled, y_binary, test_size=0.3, random_state=42
)

# Compare different kernels
kernels = ['linear', 'rbf', 'poly']
results = {}

fig, axes = plt.subplots(1, 3, figsize=(15, 5))

for idx, kernel in enumerate(kernels):
    # Train SVM
    if kernel == 'poly':
        svm = SVC(kernel=kernel, degree=3, random_state=42)
    else:
        svm = SVC(kernel=kernel, random_state=42)

    svm.fit(X_train[:, :2], y_train) # Use first 2 features for
    visualization
    score = svm.score(X_test[:, :2], y_test)
    results[kernel] = score

    # Decision boundary visualization
    ax = axes[idx]
    h = 0.02
    x_min, x_max = X_train[:, 0].min() - 1, X_train[:, 0].max() +
1
    y_min, y_max = X_train[:, 1].min() - 1, X_train[:, 1].max() +
1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))

    Z = svm.predict(xx.ravel())
    Z = np.array([1 if z == 'attack' else 0 for z in Z])
    Z = Z.reshape(xx.shape)

    ax.contourf(xx, yy, Z, alpha=0.4, cmap='RdBu')

    # Plot points
    y_numeric = np.array([1 if label == 'attack' else 0 for label
in y_train])
    scatter = ax.scatter(X_train[:, 0], X_train[:, 1],
c=y_numeric,
                      cmap='RdBu', edgecolor='black', s=50)

    # Support vectors
    if hasattr(svm, 'support_vectors_'):
        ax.scatter(svm.support_vectors_[:, 0],
svm.support_vectors_[:, 1],

```

```

        s=100, linewidth=2, facecolors='none',
edgecolors='green')

    ax.set_title(f'{kernel.capitalize()} Kernel (Acc:
{score:.3f})')
    ax.set_xlabel(feature_names[0])
    ax.set_ylabel(feature_names[1])

plt.tight_layout()
plt.show()

# Hyperparameter tuning for RBF kernel
print("\n==== SVM Hyperparameter Tuning ====")

param_grid = {
    'C': [0.1, 1, 10, 100],
    'gamma': ['scale', 'auto', 0.001, 0.01, 0.1]
}

svm = SVC(kernel='rbf')
grid_search = GridSearchCV(svm, param_grid, cv=5,
scoring='accuracy', n_jobs=-1)
grid_search.fit(X_train, y_train)

print(f"Best parameters: {grid_search.best_params_}")
print(f"Best cross-validation score:
{grid_search.best_score_:.3f}")
print(f"Test set accuracy: {grid_search.score(X_test,
y_test):.3f}")

# Confusion matrix
best_svm = grid_search.best_estimator_
y_pred = best_svm.predict(X_test)

plt.figure(figsize=(8, 6))
cm = confusion_matrix(y_test, y_pred)
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=['Normal', 'Attack'],
            yticklabels=['Normal', 'Attack'])
plt.title('SVM Confusion Matrix')
plt.ylabel('True Label')
plt.xlabel('Predicted Label')
plt.show()

self.models['svm'] = best_svm
return best_svm

def demonstrate_decision_tree(self, X, y, feature_names):
    """Demonstrate Decision Tree classification"""
    print("\n==== Decision Tree Demo ====\n")

```

```

# Split data
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42
)

# Train decision tree
dt = DecisionTreeClassifier(max_depth=4, min_samples_split=20,
random_state=42)
dt.fit(X_train, y_train)

# Evaluate
train_score = dt.score(X_train, y_train)
test_score = dt.score(X_test, y_test)
print(f"Training accuracy: {train_score:.3f}")
print(f"Test accuracy: {test_score:.3f}")

# Feature importance
importance_df = pd.DataFrame({
    'feature': feature_names,
    'importance': dt.feature_importances_
}).sort_values('importance', ascending=False)

plt.figure(figsize=(8, 6))
plt.barh(importance_df['feature'], importance_df['importance'])
plt.xlabel('Importance')
plt.title('Decision Tree Feature Importance')
plt.grid(True, alpha=0.3, axis='x')
plt.tight_layout()
plt.show()

# Visualize tree
plt.figure(figsize=(20, 10))
plot_tree(dt, feature_names=feature_names,
          class_names=np.unique(y),
          filled=True, rounded=True, fontsize=10)
plt.title('Decision Tree Visualization')
plt.show()

# Overfitting analysis
print("\n==== Overfitting Analysis ===")

max_depths = range(1, 20)
train_scores = []
test_scores = []

for depth in max_depths:
    dt_temp = DecisionTreeClassifier(max_depth=depth,
random_state=42)
    dt_temp.fit(X_train, y_train)

```

```

        train_scores.append(dt_temp.score(X_train, y_train))
        test_scores.append(dt_temp.score(X_test, y_test))

    plt.figure(figsize=(10, 6))
    plt.plot(max_depths, train_scores, 'b-', label='Training accuracy')
    plt.plot(max_depths, test_scores, 'r-', label='Test accuracy')
    plt.xlabel('Max Depth')
    plt.ylabel('Accuracy')
    plt.title('Decision Tree: Overfitting Analysis')
    plt.legend()
    plt.grid(True, alpha=0.3)
    plt.show()

    self.models['decision_tree'] = dt
    return dt

def demonstrate_random_forest(self, X, y, feature_names):
    """Demonstrate Random Forest classification"""
    print("\n==== Random Forest Demo ===\n")

    # Split data
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.3, random_state=42
    )

    # Train Random Forest
    rf = RandomForestClassifier(n_estimators=100, max_depth=10,
                               min_samples_split=5, random_state=42)
    rf.fit(X_train, y_train)

    # Evaluate
    train_score = rf.score(X_train, y_train)
    test_score = rf.score(X_test, y_test)
    print(f"Training accuracy: {train_score:.3f}")
    print(f"Test accuracy: {test_score:.3f}")

    # OOB score if available
    rf_oob = RandomForestClassifier(n_estimators=100, oob_score=True,
                                    random_state=42)
    rf_oob.fit(X_train, y_train)
    print(f"Out-of-bag score: {rf_oob.oob_score_:.3f}")

    # Feature importance comparison
    dt_importance = self.models['decision_tree'].feature_importances_
    rf_importance = rf.feature_importances_

    importance_comparison = pd.DataFrame({
        'Feature': feature_names,
        'Decision Tree': dt_importance,

```

```

        'Random Forest': rf_importance
    })

# Visualize comparison
fig, ax = plt.subplots(figsize=(10, 6))
x = np.arange(len(feature_names))
width = 0.35

ax.bar(x - width/2, importance_comparison['Decision Tree'], width,
       label='Decision Tree', alpha=0.8)
ax.bar(x + width/2, importance_comparison['Random Forest'], width,
       label='Random Forest', alpha=0.8)

ax.set_xlabel('Features')
ax.set_ylabel('Importance')
ax.set_title('Feature Importance: Decision Tree vs Random Forest')
ax.set_xticks(x)
ax.set_xticklabels(feature_names, rotation=45)
ax.legend()
ax.grid(True, alpha=0.3, axis='y')
plt.tight_layout()
plt.show()

# Number of trees analysis
print("\n==== Number of Trees Analysis ====") 169

n_trees = range(10, 201, 10)
oob_scores = []
test_scores = []

for n in n_trees:
    rf_temp = RandomForestClassifier(n_estimators=n,
oob_score=True, random_state=42)
    rf_temp.fit(X_train, y_train)
    oob_scores.append(rf_temp.oob_score_)
    test_scores.append(rf_temp.score(X_test, y_test))

plt.figure(figsize=(10, 6))
plt.plot(n_trees, oob_scores, 'b-', label='OOB Score')
plt.plot(n_trees, test_scores, 'r-', label='Test Score')
plt.xlabel('Number of Trees')
plt.ylabel('Accuracy')
plt.title('Random Forest: Performance vs Number of Trees')
plt.legend()
plt.grid(True, alpha=0.3)
plt.show()

# Classification report
y_pred = rf.predict(X_test)
print("\nClassification Report:")

```

```

        print(classification_report(y_test, y_pred))

    self.models['random_forest'] = rf
    return rf

def demonstrate_anomaly_detection(self, X, feature_names):
    """Demonstrate anomaly detection methods"""
    print("\n==== Anomaly Detection Demo ===\n")

    # Scale data
    X_scaled = self.scaler.fit_transform(X)

    # 1. Isolation Forest
    iso_forest = IsolationForest(contamination=0.1, random_state=42)
    iso_predictions = iso_forest.fit_predict(X_scaled)
    iso_scores = iso_forest.score_samples(X_scaled)

    # 2. One-Class SVM
    oc_svm = OneClassSVM(nu=0.1, kernel='rbf', gamma='auto')
    svm_predictions = oc_svm.fit_predict(X_scaled)
    svm_scores = oc_svm.score_samples(X_scaled)

    # 3. DBSCAN (outliers as anomalies)
    dbscan = DBSCAN(eps=0.5, min_samples=5)
    dbscan_labels = dbscan.fit_predict(X_scaled)
    dbscan_anomalies = dbscan_labels == -1

    # Compare methods
    print("Anomaly Detection Results:")
    print(f"Isolation Forest: {np.sum(iso_predictions == -1)} anomalies")
    print(f"One-Class SVM: {np.sum(svm_predictions == -1)} anomalies")
    print(f"DBSCAN: {np.sum(dbscan_anomalies)} anomalies")

    # Agreement between methods
    all_anomalies = ((iso_predictions == -1) &
                     (svm_predictions == -1) &
                     dbscan_anomalies)
    print(f"All methods agree: {np.sum(all_anomalies)} anomalies")

    # Visualize results
    pca = PCA(n_components=2)
    X_pca = pca.fit_transform(X_scaled)

    fig, axes = plt.subplots(2, 2, figsize=(12, 10))

    methods = [
        ('Isolation Forest', iso_predictions, iso_scores),
        ('One-Class SVM', svm_predictions, svm_scores),
        ('DBSCAN', np.where(dbscan_anomalies, -1, 1), None),
    ]

```

```

        ('Consensus', np.where(all_anomalies, -1, 1), None)
    ]

for idx, (name, predictions, scores) in enumerate(methods):
    ax = axes[idx // 2, idx % 2]

    if scores is not None:
        scatter = ax.scatter(X_pca[:, 0], X_pca[:, 1],
                             c=scores, cmap='YlOrRd', alpha=0.6)
        plt.colorbar(scatter, ax=ax, label='Anomaly Score')
    else:
        colors = ['red' if p == -1 else 'blue' for p in
predictions]
        ax.scatter(X_pca[:, 0], X_pca[:, 1], c=colors, alpha=0.6)

    # Mark anomalies
    anomaly_mask = predictions == -1
    ax.scatter(X_pca[anomaly_mask, 0], X_pca[anomaly_mask, 1],
               marker='x', s=100, c='black', linewidths=3)

    ax.set_title(f'{name} ({np.sum(anomaly_mask)} anomalies)')
    ax.set_xlabel('PC1')
    ax.set_ylabel('PC2')
    ax.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

# Analyze detected anomalies
print("\n==== Anomaly Analysis ===")
consensus_indices = np.where(all_anomalies)[0]

if len(consensus_indices) > 0:
    print(f"\nConsensus anomalies (indices:
{consensus_indices[:5]}...)")
    anomaly_data = X[consensus_indices]
    normal_data = X[~all_anomalies]

    print("\nFeature comparison (Anomaly vs Normal):")
    for i, feature in enumerate(feature_names):
        anomaly_mean = anomaly_data[:, i].mean()
        normal_mean = normal_data[:, i].mean()
        ratio = anomaly_mean / normal_mean if normal_mean > 0 else
np.inf
        print(f"{feature}: Anomaly={anomaly_mean:.2f}, Normal=
{normal_mean:.2f}, Ratio={ratio:.2f}")

# Main demonstration
def main():
    """Run comprehensive ML security demonstrations"""

```

```

toolkit = SecurityMLToolkit()

# Generate network security dataset
X_network, y_network, features_network =
toolkit.generate_security_dataset('network')
print(f"Network dataset: {X_network.shape[0]} samples,
{X_network.shape[1]} features")
print(f"Classes: {np.unique(y_network, return_counts=True)}")

# 1. K-Means demonstration
kmeans_labels = toolkit.demonstrate_kmeans(X_network,
features_network)

# 2. DBSCAN demonstration
dbscan_labels = toolkit.demonstrate_dbSCAN(X_network,
features_network)

# 3. PCA demonstration
X_pca = toolkit.demonstrate_pca(X_network, y_network,
features_network)

# 4. SVM demonstration
svm_model = toolkit.demonstrate_svm(X_network, y_network,
features_network) 172

# 5. Decision Tree demonstration
dt_model = toolkit.demonstrate_decision_tree(X_network, y_network,
features_network)

# 6. Random Forest demonstration
rf_model = toolkit.demonstrate_random_forest(X_network, y_network,
features_network)

# 7. Anomaly Detection demonstration
toolkit.demonstrate_anomaly_detection(X_network, features_network)

# Generate user behavior dataset
print("\n" + "="*50 + "\n")
print("== Switching to User Behavior Analysis ==")
X_user, y_user, features_user =
toolkit.generate_security_dataset('user_behavior')
print(f"User behavior dataset: {X_user.shape[0]} samples,
{X_user.shape[1]} features")
print(f"Classes: {np.unique(y_user, return_counts=True)}")

# Apply anomaly detection to user behavior
toolkit.demonstrate_anomaly_detection(X_user, features_user)

```

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

---

## **Exercise 1: Multi-Algorithm Comparison**

Create a system that:

1. Loads a security dataset with labeled anomalies
2. Applies K-Means, DBSCAN, and GMM clustering
3. Compares their ability to separate normal from anomalous
4. Visualizes results using dimensionality reduction
5. Provides recommendations for which algorithm to use

## **Exercise 2: Adaptive Anomaly Detection**

Build an anomaly detection pipeline that:

1. Combines multiple algorithms (Isolation Forest, One-Class SVM, Autoencoder)
2. Learns normal behavior patterns over time
3. Adapts thresholds based on feedback
4. Provides confidence scores for detections
5. Handles concept drift in data

## **Exercise 3: Real-Time Classification System**

174

Implement a security event classifier that:

1. Uses Random Forest for initial classification
2. Falls back to SVM for uncertain cases
3. Updates incrementally with new data
4. Monitors feature importance changes
5. Alerts on classification pattern shifts

## **Exercise 4: Hierarchical Security Analysis**

Create a hierarchical clustering system that:

1. Groups security events at multiple granularities
2. Identifies event taxonomies automatically
3. Detects new event types as they emerge
4. Visualizes the security event hierarchy
5. Provides drill-down capabilities

This section covered fundamental machine learning algorithms:

## Key Algorithms Mastered:

- **Clustering:** K-Means (spherical), DBSCAN (density-based), Hierarchical (taxonomy)
- **Classification:** KNN (local), SVM (margin-based), Trees (rule-based), Forests (ensemble)
- **Dimensionality Reduction:** PCA for visualization and feature extraction
- **Anomaly Detection:** Multiple approaches for outlier identification

## Critical Insights:

- **No Free Lunch:** No single algorithm works best for all problems
- **Algorithm Selection:** Choose based on data characteristics and requirements
- **Ensemble Power:** Combining models often outperforms individuals
- **Preprocessing Matters:** Scaling and feature engineering are crucial
- **Validation is Key:** Always use proper train/test splits and cross-validation

## Decision Framework:

1. **Understand your data:** Shape, size, dimensionality, noise level
2. **Define success metrics:** Accuracy, interpretability, speed, scalability
3. **Start simple:** Try linear/simple models first
4. **Iterate and combine:** Use ensembles and stacking
5. **Monitor in production:** Models degrade over time

175

## Next Steps:

In Section 4, we'll dive into deep learning, building on these foundations to create neural networks that can learn even more complex patterns in security data.

# Machine Learning Training Documentation

## Section 4: Deep Learning Essentials

### Overview

This section introduces neural networks and deep learning fundamentals. We'll build from simple linear models to complex multi-layer architectures, understanding the mathematical foundations that power modern AI systems. By the end, you'll be able to design, train, and deploy neural networks for security applications.

### Learning Objectives

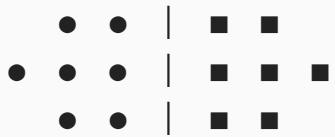
By the end of this section, you will be able to:

- Understand the mathematical foundations of neural networks
- Implement gradient descent and backpropagation from scratch
- Choose appropriate activation functions and architectures
- Apply regularization techniques to prevent overfitting
- Build multi-class classifiers for security applications
- Deploy real-time neural network models

# The Limitations of Linear Models

Linear models can only learn linear decision boundaries:

Linear Separable



✓ Can separate

Non-Linear Problem



✗ Cannot separate

# The Perceptron: Building Block of Neural Networks

## Mathematical Foundation

A perceptron computes:

$$z = w_1x_1 + w_2x_2 + \dots + w_nx_n + b = w^T x + b$$
$$y = \text{activation}(z)$$

## Perceptron Components

177

| Component  | Symbol   | Purpose              | Dimension |
|------------|----------|----------------------|-----------|
| Input      | x        | Features             | (n,)      |
| Weights    | w        | Feature importance   | (n,)      |
| Bias       | b        | Threshold adjustment | scalar    |
| Activation | $\sigma$ | Non-linearity        | function  |
| Output     | y        | Prediction           | scalar    |

## Memory Trick: "WIBA"

Weights multiply Inputs, add Bias, apply Activation

# Why Activation Functions?

Without activation functions, stacking linear layers still produces linear models:

$$\text{Linear}(\text{Linear}(x)) = \text{Linear}(x)$$

## Common Activation Functions

| Function          | Formula                                    | Range  | Use Case           | Pros                     | Cons               |
|-------------------|--|--------|--------------------|--------------------------|--------------------|
| <b>Sigmoid</b>    | $\sigma(x) = 1/(1+e^{-x})$                 | (0,1)  | Binary output      | Smooth gradient          | Vanishing gradient |
| <b>Tanh</b>       | $\tanh(x) = (e^x - e^{-x})/(e^x + e^{-x})$ | (-1,1) | Hidden layers      | Zero-centered            | Vanishing gradient |
| <b>ReLU</b>       | $\max(0,x)$                                | [0,∞)  | Hidden layers      | No vanishing gradient    | Dead neurons       |
| <b>Leaky ReLU</b> | $\max(0.01x, x)$                           | (-∞,∞) | Hidden layers      | Fixes dead ReLU          | Extra parameter    |
| <b>Softmax</b>    | $e^{xi}/\sum e^{xj}$                       | (0,1)  | Multi-class output | Probability distribution | Only for output    |

178

## Visual Representation

```
import numpy as np
import matplotlib.pyplot as plt

# Visualize activation functions
x = np.linspace(-5, 5, 100)

fig, axes = plt.subplots(2, 3, figsize=(15, 10))
axes = axes.ravel()

# Sigmoid
sigmoid = 1 / (1 + np.exp(-x))
axes[0].plot(x, sigmoid, 'b-', linewidth=2)
axes[0].set_title('Sigmoid:  $\sigma(x) = 1/(1+e^{-x})$ ', fontsize=12)
axes[0].grid(True, alpha=0.3)
axes[0].axhline(y=0.5, color='r', linestyle='--', alpha=0.5)
axes[0].text(0, 0.5, 'y=0.5', color='r')

# Tanh
tanh = np.tanh(x)
axes[1].plot(x, tanh, 'g-', linewidth=2)
axes[1].set_title('Tanh:  $(e^x - e^{-x})/(e^x + e^{-x})$ ', fontsize=12)
```

```

axes[1].grid(True, alpha=0.3)
axes[1].axhline(y=0, color='r', linestyle='--', alpha=0.5)

# ReLU
relu = np.maximum(0, x)
axes[2].plot(x, relu, 'r-', linewidth=2)
axes[2].set_title('ReLU: max(0,x)', fontsize=12)
axes[2].grid(True, alpha=0.3)
axes[2].axvline(x=0, color='b', linestyle='--', alpha=0.5)

# Leaky ReLU
leaky_relu = np.where(x > 0, x, 0.01 * x)
axes[3].plot(x, leaky_relu, 'm-', linewidth=2)
axes[3].set_title('Leaky ReLU: max(0.01x,x)', fontsize=12)
axes[3].grid(True, alpha=0.3)

# ELU
elu = np.where(x > 0, x, np.exp(x) - 1)
axes[4].plot(x, elu, 'c-', linewidth=2)
axes[4].set_title('ELU: x if x>0 else e^x-1', fontsize=12)
axes[4].grid(True, alpha=0.3)

# Softmax (for 3 classes)
softmax_input = np.array([x, x-1, x+1]).T
softmax_output = np.exp(softmax_input) / np.exp(softmax_input).sum(axis=1, keepdims=True)
axes[5].plot(x, softmax_output[:, 0], 'b-', label='Class 0', linewidth=2)
axes[5].plot(x, softmax_output[:, 1], 'g-', label='Class 1', linewidth=2)
axes[5].plot(x, softmax_output[:, 2], 'r-', label='Class 2', linewidth=2)
axes[5].set_title('Softmax: e^x_i / \sum e^x_j', fontsize=12)
axes[5].legend()
axes[5].grid(True, alpha=0.3)

for ax in axes:
    ax.set_xlabel('x')
    ax.set_ylabel('f(x)')

plt.tight_layout()
plt.show()

```

179

## Activation Function Selection Guide

Input Layer → No activation (raw features)

↓

Hidden Layers → ReLU (default choice)

↓

→ Leaky ReLU (if dead neurons)

↓

→ ELU (if need smoothness)

↓

Output Layer → Sigmoid (binary classification)

→ Softmax (multi-class)

→ Linear (regression)

---

# Purpose of Loss Functions

Loss functions measure how wrong our predictions are:

Perfect Prediction: Loss = 0

Bad Prediction: Loss > 0

## Common Loss Functions

| Loss Function             | Formula  | Use Case              | Properties                   |
|---------------------------|--|-----------------------|------------------------------|
| MSE                       | $(1/n)\sum(y - \hat{y})^2$                                   | Regression            | Penalizes large errors       |
| MAE                       | $(1/n)\sum y - \hat{y} $                                     | $y - \hat{y}$         |                              |
| Binary Cross-entropy      | $-\sum(y \cdot \log(\hat{y}) + (1-y) \cdot \log(1-\hat{y}))$ | Binary classification | Probabilistic interpretation |
| Categorical Cross-entropy | $-\sum \sum y_{ij} \cdot \log(\hat{y}_{ij})$                 | Multi-class           | For one-hot encoded labels   |
| Hinge Loss                | $\max(0, 1 - y \cdot \hat{y})$                               | SVM                   | Margin-based                 |

## Visual Comparison of Loss Functions

181

```
# Visualize different loss functions
y_true = 1 # True label for binary classification
y_pred = np.linspace(0, 1, 100)

fig, axes = plt.subplots(2, 2, figsize=(12, 10))

# Binary cross-entropy
bce = -(y_true * np.log(y_pred + 1e-7) + (1-y_true) * np.log(1-y_pred + 1e-7))
axes[0, 0].plot(y_pred, bce, 'b-', linewidth=2)
axes[0, 0].set_title('Binary Cross-Entropy (y_true=1)')
axes[0, 0].set_xlabel('Predicted Probability')
axes[0, 0].set_ylabel('Loss')
axes[0, 0].grid(True, alpha=0.3)

# MSE for regression
y_true_reg = 0.7
mse = (y_true_reg - y_pred) ** 2
axes[0, 1].plot(y_pred, mse, 'g-', linewidth=2)
axes[0, 1].set_title('Mean Squared Error (y_true=0.7)')
axes[0, 1].set_xlabel('Predicted Value')
axes[0, 1].set_ylabel('Loss')
axes[0, 1].grid(True, alpha=0.3)
```

```

# MAE for regression
mae = np.abs(y_true_reg - y_pred)
axes[1, 0].plot(y_pred, mae, 'r-', linewidth=2)
axes[1, 0].set_title('Mean Absolute Error (y_true=0.7)')
axes[1, 0].set_xlabel('Predicted Value')
axes[1, 0].set_ylabel('Loss')
axes[1, 0].grid(True, alpha=0.3)

# Comparison
axes[1, 1].plot(y_pred, bce/bce.max(), 'b-', label='BCE (normalized)', linewidth=2)
axes[1, 1].plot(y_pred, mse/mse.max(), 'g-', label='MSE (normalized)', linewidth=2)
axes[1, 1].plot(y_pred, mae/mae.max(), 'r-', label='MAE (normalized)', linewidth=2)
axes[1, 1].set_title('Loss Function Comparison')
axes[1, 1].set_xlabel('Predicted Value')
axes[1, 1].set_ylabel('Normalized Loss')
axes[1, 1].legend()
axes[1, 1].grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

```

182

## Loss Function Selection Matrix

| Problem Type            | Output Activation | Loss Function             | Why?                               |
|-------------------------|-------------------|---------------------------|------------------------------------|
| Binary Classification   | Sigmoid           | Binary Cross-entropy      | Log loss for probabilities         |
| Multi-class (exclusive) | Softmax           | Categorical Cross-entropy | Mutual exclusivity                 |
| Multi-label             | Sigmoid           | Binary Cross-entropy      | Independent probabilities          |
| Regression              | Linear            | MSE                       | Differentiable, penalizes outliers |
| Robust Regression       | Linear            | MAE or Huber              | Less sensitive to outliers         |

# Gradient Descent Intuition

Imagine finding the lowest point in a valley while blindfolded:

```
Current Position  
↓  
Calculate Slope  
↓  
Step Downhill  
↓  
Repeat Until Flat
```

## Mathematical Foundation

### Gradient Descent Update Rule:

$$\theta_{\text{new}} = \theta_{\text{old}} - \alpha \times \nabla L(\theta)$$

Where:

- $\theta$  = parameters (weights, biases)
- $\alpha$  = learning rate
- $\nabla L$  = gradient of loss

183

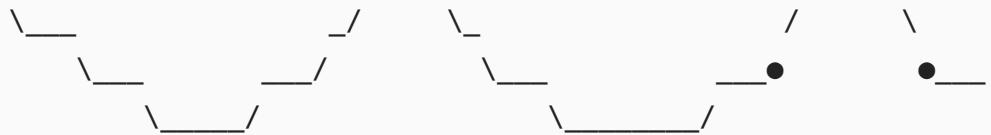
## Types of Gradient Descent

| Type          | Batch Size   | Update Frequency | Pros                          | Cons                   |
|---------------|--------------|------------------|-------------------------------|------------------------|
| Batch GD      | Full dataset | Once per epoch   | Stable                        | Slow, memory intensive |
| Stochastic GD | 1 sample     | Every sample     | Fast, can escape local minima | Noisy                  |
| Mini-batch GD | 16-512       | Every batch      | Balance of both               | Hyperparameter tuning  |

## Learning Rate Effects

Too Small ( $\alpha=0.0001$ )      Just Right ( $\alpha=0.01$ )      Too Large ( $\alpha=1.0$ )





Slow convergence

Smooth convergence

Oscillation/Divergence

## Backpropagation Algorithm

The chain rule in action:

$$\text{Output Layer: } \frac{\partial L}{\partial w_3} = \frac{\partial L}{\partial y} \times \frac{\partial y}{\partial z_3} \times \frac{\partial z_3}{\partial w_3}$$

↑

$$\text{Hidden Layer 2: } \frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial y} \times \frac{\partial y}{\partial z_3} \times \frac{\partial z_3}{\partial a_2} \times \frac{\partial a_2}{\partial z_2} \times \frac{\partial z_2}{\partial w_2}$$

↑

$$\text{Hidden Layer 1: } \frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y} \times \dots \times \frac{\partial z_1}{\partial w_1}$$

## Memory Trick: "FLEB"

Forward pass → calculate Loss → Error backwards → update Bias & weights

# Network Architecture Patterns

| Architecture              | Layers                     | Parameters | Use Case           |
|---------------------------|----------------------------|------------|--------------------|
| <b>Shallow &amp; Wide</b> | 2-3 layers, many neurons   | High       | Tabular data       |
| <b>Deep &amp; Narrow</b>  | Many layers, fewer neurons | Moderate   | Complex patterns   |
| <b>Funnel</b>             | Decreasing width           | Moderate   | Feature extraction |
| <b>Hourglass</b>          | Narrow middle              | High       | Autoencoders       |

## Implementation: Building Blocks

```
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# Generate security-relevant dataset
X, y = make_classification(
    n_samples=1000,
    n_features=20,
    n_informative=15,
    n_redundant=5,
    n_classes=2,
    flip_y=0.1,
    random_state=42
)

# Split and scale
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Build a simple neural network with visualization
def build_and_visualize_network(architecture_type='standard'):
    """Build different neural network architectures"""

    if architecture_type == 'standard':
        model = keras.Sequential([
            layers.Input(shape=(20,)),
```

```

        layers.Dense(64, activation='relu', name='hidden1'),
        layers.Dense(32, activation='relu', name='hidden2'),
        layers.Dense(16, activation='relu', name='hidden3'),
        layers.Dense(1, activation='sigmoid', name='output')
    ])

elif architecture_type == 'funnel':
    model = keras.Sequential([
        layers.Input(shape=(20,)),
        layers.Dense(128, activation='relu', name='wide'),
        layers.Dense(64, activation='relu', name='medium'),
        layers.Dense(32, activation='relu', name='narrow'),
        layers.Dense(1, activation='sigmoid', name='output')
    ])

elif architecture_type == 'regularized':
    model = keras.Sequential([
        layers.Input(shape=(20,)),
        layers.Dense(64, activation='relu'),
        layers.Dropout(0.3),
        layers.Dense(32, activation='relu'),
        layers.BatchNormalization(),
        layers.Dense(16, activation='relu'),
        layers.Dropout(0.2),
        layers.Dense(1, activation='sigmoid')
    ])

    return model

# Visualize different architectures
fig, axes = plt.subplots(1, 3, figsize=(15, 5))

architectures = ['standard', 'funnel', 'regularized']
titles = ['Standard Network', 'Funnel Architecture', 'Regularized Network']

for idx, (arch, title) in enumerate(zip(architectures, titles)):
    model = build_and_visualize_network(arch)

    # Get layer information
    layer_sizes = [20] # Input size
    layer_names = ['Input']

    for layer in model.layers:
        if hasattr(layer, 'units'):
            layer_sizes.append(layer.units)
            layer_names.append(layer.name if hasattr(layer, 'name') else
'Layer')

    # Draw network

```

```

ax = axes[idx]
layer_positions = np.linspace(0, 1, len(layer_sizes))

# Draw neurons and connections
for i in range(len(layer_sizes) - 1):
    y1 = np.linspace(0.1, 0.9, min(layer_sizes[i], 10))
    y2 = np.linspace(0.1, 0.9, min(layer_sizes[i+1], 10))

    # Draw connections
    for j in range(len(y1)):
        for k in range(len(y2)):
            ax.plot([layer_positions[i], layer_positions[i+1]],
                    [y1[j], y2[k]], 'gray', alpha=0.1, linewidth=0.5)

    # Draw neurons
    ax.scatter([layer_positions[i]] * len(y1), y1, s=300, c='blue',
               edgecolor='black', linewidth=2, zorder=10)

# Draw output layer
ax.scatter([layer_positions[-1]], [0.5], s=300, c='red',
           edgecolor='black', linewidth=2, zorder=10)

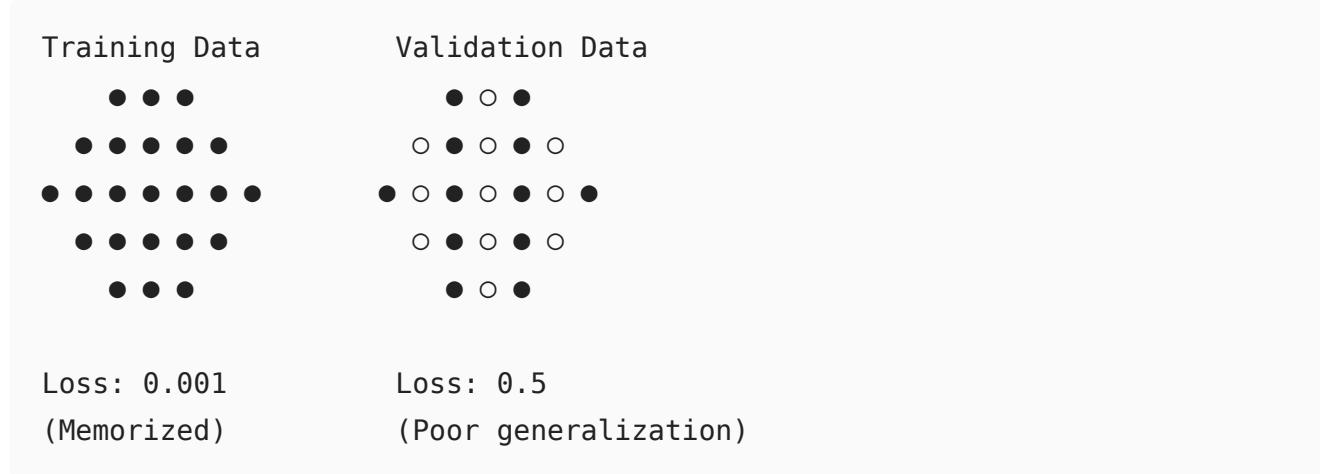
# Labels
for i, (pos, name) in enumerate(zip(layer_positions, layer_names)):
    ax.text(pos, -0.1, name, ha='center', fontsize=10)
    if i < len(layer_sizes):
        ax.text(pos, 1.05, f'{layer_sizes[i]} units', ha='center',
                fontsize=8)

    ax.set_xlim(-0.1, 1.1)
    ax.set_ylim(-0.2, 1.2)
    ax.axis('off')
    ax.set_title(title, fontsize=12, fontweight='bold')

plt.tight_layout()
plt.show()

```

# The Overfitting Problem



## Regularization Methods

| Method                     | How it Works                  | When to Use     | Implementation              |
|----------------------------|-------------------------------|-----------------|-----------------------------|
| <b>L1 Regularization</b>   | Adds $\Sigma w$               | w               | to loss                     |
| <b>L2 Regularization</b>   | Adds $\Sigma w^2$ to loss     | Smooth weights  | kernel_regularizer=l2(0.01) |
| <b>Dropout</b>             | Randomly zeros neurons        | General purpose | Dropout(rate=0.5)           |
| <b>Early Stopping</b>      | Stop when validation plateaus | Always          | EarlyStopping(patience=10)  |
| <b>Batch Normalization</b> | Normalize layer inputs        | Deep networks   | BatchNormalization()        |
| <b>Data Augmentation</b>   | Create synthetic examples     | Limited data    | Custom transforms           |

188

## Regularization Comparison

```
# Compare regularization techniques
def train_with_regularization(X_train, y_train, X_val, y_val,
reg_type='none'):
    """Train models with different regularization"""

    if reg_type == 'none':
        model = keras.Sequential([
            layers.Dense(128, activation='relu', input_shape=
(X_train.shape[1],)),
            layers.Dense(64, activation='relu'),
```

```

        layers.Dense(32, activation='relu'),
        layers.Dense(1, activation='sigmoid')
    ])

elif reg_type == 'l2':
    model = keras.Sequential([
        layers.Dense(128, activation='relu', input_shape=
(X_train.shape[1],),
                    kernel_regularizer=keras.regularizers.l2(0.01)),
        layers.Dense(64, activation='relu',
                    kernel_regularizer=keras.regularizers.l2(0.01)),
        layers.Dense(32, activation='relu',
                    kernel_regularizer=keras.regularizers.l2(0.01)),
        layers.Dense(1, activation='sigmoid')
    ])

elif reg_type == 'dropout':
    model = keras.Sequential([
        layers.Dense(128, activation='relu', input_shape=
(X_train.shape[1],)),
        layers.Dropout(0.5),
        layers.Dense(64, activation='relu'),
        layers.Dropout(0.3),
        layers.Dense(32, activation='relu'),
        layers.Dropout(0.2),
        layers.Dense(1, activation='sigmoid')
    ])

elif reg_type == 'combined':
    model = keras.Sequential([
        layers.Dense(128, activation='relu', input_shape=
(X_train.shape[1],),
                    kernel_regularizer=keras.regularizers.l2(0.01)),
        layers.BatchNormalization(),
        layers.Dropout(0.3),
        layers.Dense(64, activation='relu',
                    kernel_regularizer=keras.regularizers.l2(0.01)),
        layers.BatchNormalization(),
        layers.Dropout(0.2),
        layers.Dense(32, activation='relu'),
        layers.Dense(1, activation='sigmoid')
    ])

model.compile(
    optimizer='adam',
    loss='binary_crossentropy',
    metrics=['accuracy']
)

# Train with early stopping

```

```

    early_stop = keras.callbacks.EarlyStopping(
        monitor='val_loss',
        patience=10,
        restore_best_weights=True
    )

    history = model.fit(
        X_train, y_train,
        validation_data=(X_val, y_val),
        epochs=100,
        batch_size=32,
        callbacks=[early_stop],
        verbose=0
    )

    return model, history
}

# Train models with different regularization
X_train_split, X_val_split, y_train_split, y_val_split = train_test_split(
    X_train_scaled, y_train, test_size=0.2, random_state=42
)

reg_types = ['none', 'l2', 'dropout', 'combined']
histories = {}

for reg_type in reg_types:
    print(f"Training with {reg_type} regularization...")
    _, history = train_with_regularization(
        X_train_split, y_train_split, X_val_split, y_val_split, reg_type
    )
    histories[reg_type] = history

# Visualize regularization effects
fig, axes = plt.subplots(2, 2, figsize=(12, 10))
axes = axes.ravel()

for idx, (reg_type, history) in enumerate(histories.items()):
    ax = axes[idx]

    # Plot training and validation loss
    ax.plot(history.history['loss'], label='Training Loss', linewidth=2)
    ax.plot(history.history['val_loss'], label='Validation Loss',
            linewidth=2)

    # Mark overfitting region
    train_loss = history.history['loss']
    val_loss = history.history['val_loss']
    divergence_point = np.where(np.array(val_loss) > np.array(train_loss)
* 1.2)[0]

```

```
if len(divergence_point) > 0:
    ax.axvspan(divergence_point[0], len(train_loss), alpha=0.3,
color='red',
               label='Overfitting Region')

ax.set_title(f'Regularization: {reg_type.capitalize()}', fontsize=12)
ax.set_xlabel('Epoch')
ax.set_ylabel('Loss')
ax.legend()
ax.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()
```

# One-hot Encoding

Transform categorical labels into binary vectors:

| Label  | One-hot Encoding |
|--------|------------------|
| "HTTP" | → [1, 0, 0, 0]   |
| "SSH"  | → [0, 1, 0, 0]   |
| "FTP"  | → [0, 0, 1, 0]   |
| "DNS"  | → [0, 0, 0, 1]   |

# Softmax Activation

Converts raw scores to probabilities:

$$\text{Softmax}(z_i) = e^{z_i} / \sum_j e^{z_j}$$

Properties:

- Sum to 1.0
- All values in (0,1)
- Differentiable

192

# Multi-class Loss Functions

| Loss Function                    | When to Use        | Formula                              |
|----------------------------------|--------------------|--------------------------------------|
| Categorical Cross-entropy        | One-hot labels     | $-\sum_i y_i \log(\hat{y}_i)$        |
| Sparse Categorical Cross-entropy | Integer labels     | $-\log(\hat{y}[\text{true\_class}])$ |
| Focal Loss                       | Imbalanced classes | $-(1-p_t)^\gamma \log(p_t)$          |

# Real-time Protocol Classification

```
# Generate multi-class network protocol data
from sklearn.datasets import make_classification

# Create protocol classification dataset
X_proto, y_proto = make_classification(
    n_samples=5000,
    n_features=15, # packet size, timing, port, flags, etc.
    n_informative=12,
    n_redundant=3,
    n_classes=5, # HTTP, SSH, FTP, DNS, OTHER
    n_clusters_per_class=2,
```

```

        flip_y=0.05,
        random_state=42
    )

# Split data
X_train_proto, X_test_proto, y_train_proto, y_test_proto =
train_test_split(
    X_proto, y_proto, test_size=0.2, random_state=42
)

# Scale features
scaler_proto = StandardScaler()
X_train_proto_scaled = scaler_proto.fit_transform(X_train_proto)
X_test_proto_scaled = scaler_proto.transform(X_test_proto)

# Build multi-class classifier
def build_protocol_classifier():
    model = keras.Sequential([
        layers.Input(shape=(15,)),
        layers.Dense(64, activation='relu'),
        layers.BatchNormalization(),
        layers.Dropout(0.3),
        layers.Dense(32, activation='relu'),
        layers.BatchNormalization(),
        layers.Dropout(0.2),
        layers.Dense(16, activation='relu'),
        layers.Dense(5, activation='softmax') # 5 classes
    ])

    model.compile(
        optimizer='adam',
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy']
    )

    return model

# Train protocol classifier
protocol_model = build_protocol_classifier()

history_proto = protocol_model.fit(
    X_train_proto_scaled, y_train_proto,
    validation_split=0.2,
    epochs=50,
    batch_size=32,
    verbose=0
)

# Evaluate model
test_loss, test_accuracy = protocol_model.evaluate(

```

```
X_test_proto_scaled, y_test_proto, verbose=0
)

print(f"Protocol Classification Accuracy: {test_accuracy:.3f}")

# Visualize predictions
predictions = protocol_model.predict(X_test_proto_scaled[:10])
protocol_names = ['HTTP', 'SSH', 'FTP', 'DNS', 'OTHER']

fig, axes = plt.subplots(2, 5, figsize=(15, 6))
axes = axes.ravel()

for i in range(10):
    ax = axes[i]
    probs = predictions[i]
    true_class = y_test_proto[i]

    # Bar plot of probabilities
    bars = ax.bar(range(5), probs)
    bars[true_class].set_color('green')

    # Highlight prediction
    pred_class = np.argmax(probs)
    if pred_class != true_class:
        bars[pred_class].set_color('red')

    ax.set_ylim(0, 1)
    ax.set_xticks(range(5))
    ax.set_xticklabels(protocol_names, rotation=45)
    ax.set_title(f'True: {protocol_names[true_class]}', fontsize=10)
    ax.grid(True, alpha=0.3, axis='y')

plt.suptitle('Protocol Classification Predictions', fontsize=14)
plt.tight_layout()
plt.show()
```

# Performance Optimization

| Technique              | Speed Improvement | Trade-off            |
|------------------------|-------------------|----------------------|
| Quantization           | 2-4x              | Slight accuracy loss |
| Pruning                | 2-10x             | Need retraining      |
| Knowledge Distillation | 2-5x              | Training complexity  |
| ONNX Runtime           | 1.5-3x            | Conversion overhead  |
| Batch Processing       | 5-20x             | Latency increase     |

## Memory Patterns

|                  |                  |
|------------------|------------------|
| Standard Model:  | Quantized Model: |
| 32-bit floats    | 8-bit integers   |
| └ 4 bytes/weight | └ 1 byte/weight  |
| └ 100MB model    | └ 25MB model     |
| └ Slow mobile    | └ Fast mobile    |

## Production Checklist

195

- Model versioning:** Track model versions
- Input validation:** Check data ranges
- Error handling:** Graceful failure
- Monitoring:** Track predictions
- A/B testing:** Compare models
- Rollback plan:** Quick reversion
- API rate limiting:** Prevent abuse
- Security:** Input sanitization

## 4.9 Advanced Autoencoder Loss Functions

### Overview

Autoencoders require specialized loss functions to effectively learn compressed representations. The choice of loss function significantly impacts reconstruction quality, training stability, and the model's ability to handle different data types.

### Detailed Loss Function Comparison for Autoencoders

| Loss Function        | Formula  | Use Case                | Advantages                  | Disadvantages                                 |
|----------------------|--|-------------------------|-----------------------------|---|
| MSE                  | $(1/n)\sum(x - \hat{x})^2$                                   | Continuous data, images | Simple, smooth gradients    | Sensitive to outliers, blurry reconstructions |
| MAE                  | $(1/n)\sum x - \hat{x} $                                     | Robust reconstruction   | Outlier resistant           | Non-smooth gradients, slower convergence      |
| Binary Cross-Entropy | $-\sum[x \cdot \log(\hat{x}) + (1-x) \cdot \log(1-\hat{x})]$ | Binary/normalized data  | Better for probabilities    | Requires [0,1] range                          |
| Huber Loss           | See below  | Mixed scenarios         | Combines MSE & MAE benefits | Requires $\delta$ tuning                      |
| RMSE                 | $\sqrt{(1/n)\sum(x - \hat{x})^2}$                            | When scale matters      | Interpretable units         | Same as MSE for optimization                  |

## Huber Loss - The Hybrid Approach

Huber loss combines the best of MSE and MAE:

196

```
def huber_loss(y_true, y_pred, delta=1.0):
    """
    Huber loss: quadratic for small errors, linear for large errors

    L = {
        0.5 * (y_true - y_pred)**2           for |y_true - y_pred| <= delta
        delta * |y_true - y_pred| - 0.5 * delta**2   otherwise
    }
    """

    error = y_true - y_pred
    is_small_error = tf.abs(error) <= delta

    squared_loss = 0.5 * tf.square(error)
    linear_loss = delta * tf.abs(error) - 0.5 * tf.square(delta)

    return tf.where(is_small_error, squared_loss, linear_loss)
```

## Vanishing Gradient Analysis

### How Different Loss Functions Handle Gradient Vanishing

```
import numpy as np
import matplotlib.pyplot as plt
```

```

import tensorflow as tf

def visualize_gradient_behavior():
    """Compare gradient behavior near convergence"""

    # Create error range from very small to large
    errors = np.logspace(-6, 1, 1000) # 10^-6 to 10^1

    # Calculate gradients for different loss functions
    # MSE gradient: 2(ŷ - x)
    mse_grad = 2 * errors

    # MAE gradient: sign(ŷ - x)
    mae_grad = np.ones_like(errors)

    # Huber gradient (delta=1.0)
    delta = 1.0
    huber_grad = np.where(errors <= delta, errors, delta *
    np.sign(errors))

    # BCE gradient approximation (for x=0.5, ŷ varying)
    x_true = 0.5
    x_pred = 0.5 + errors/10 # Small deviations from true value
    x_pred = np.clip(x_pred, 1e-7, 1-1e-7) # Avoid log(0)
    bce_grad = (x_pred - x_true) / (x_pred * (1 - x_pred))

    # Plotting
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 6))

    # Log scale plot
    ax1.loglog(errors, mse_grad, label='MSE:  $2\epsilon$ ', linewidth=2)
    ax1.loglog(errors, mae_grad, label='MAE:  $1$ ', linewidth=2)
    ax1.loglog(errors, huber_grad, label=f'Huber ( $\delta={\delta}$ )',
    linewidth=2, linestyle='--')
    ax1.axvline(x=1e-3, color='red', linestyle=':', alpha=0.5,
    label='Typical convergence')
    ax1.set_xlabel('Error Magnitude')
    ax1.set_ylabel('Gradient Magnitude')
    ax1.set_title('Gradient Behavior (Log Scale)')
    ax1.legend()
    ax1.grid(True, alpha=0.3)

    # Zoomed view near zero
    small_errors = errors[errors < 0.1]
    ax2.plot(small_errors, 2 * small_errors, label='MSE', linewidth=2)
    ax2.plot(small_errors, np.ones_like(small_errors), label='MAE',
    linewidth=2)
    ax2.plot(small_errors, small_errors, label='Huber', linewidth=2,
    linestyle='--')
    ax2.set_xlabel('Error Magnitude')

```

```

        ax2.set_ylabel('Gradient Magnitude')
        ax2.set_title('Gradient Behavior Near Zero')
        ax2.legend()
        ax2.grid(True, alpha=0.3)

    plt.tight_layout()
    plt.show()

# Visualize the gradients
visualize_gradient_behavior()

```

## Practical Implementation Guide

### 1. Adaptive Loss Selection

```

class AdaptiveAutoencoderLoss(tf.keras.losses.Loss):
    """
    Automatically switches between loss functions based on training
    progress
    """

    def __init__(self, initial_loss='mse', switch_epoch=50):
        super().__init__()
        self.initial_loss = initial_loss
        self.switch_epoch = switch_epoch
        self.current_epoch = 0

    def call(self, y_true, y_pred):
        # Start with MSE for stable gradients
        if self.current_epoch < self.switch_epoch:
            return tf.reduce_mean(tf.square(y_true - y_pred))
        else:
            # Switch to Huber for fine-tuning
            return tf.reduce_mean(
                tf.keras.losses.huber(y_true, y_pred, delta=1.0)
            )

```

198

### 2. Combined Loss for Different Data Types

```

def mixed_reconstruction_loss(x_true, x_pred,
                             continuous_mask, binary_mask,
                             alpha=0.5):
    """
    Combine MSE for continuous features and BCE for binary features
    """

    # MSE for continuous features
    continuous_loss = tf.reduce_mean(
        tf.square((x_true - x_pred) * continuous_mask)

```

```

        )

# BCE for binary features
binary_loss = tf.reduce_mean(
    tf.keras.losses.binary_crossentropy(
        x_true * binary_mask,
        x_pred * binary_mask,
        from_logits=False
    )
)

return alpha * continuous_loss + (1 - alpha) * binary_loss

```

## Strategies to Combat Vanishing Gradients

### 1. Gradient Clipping

```
optimizer = tf.keras.optimizers.Adam(learning_rate=0.001, clipnorm=1.0)
```

### 2. Learning Rate Scheduling

```

def autoencoder_lr_schedule(epoch, lr):
    """Adaptive learning rate for autoencoder training"""
    if epoch < 10:
        return lr
    elif epoch < 50:
        return lr * 0.1
    else:
        # Very small LR when loss approaches zero
        return lr * 0.01

lr_callback =
tf.keras.callbacks.LearningRateScheduler(autoencoder_lr_schedule)

```

199

### 3. Loss Scaling

```

class ScaledMSE(tf.keras.losses.Loss):
    """MSE with dynamic scaling to prevent vanishing gradients"""
    def __init__(self, scale_factor=1000):
        super().__init__()
        self.scale_factor = scale_factor

    def call(self, y_true, y_pred):
        # Scale up the loss to maintain gradient magnitude
        mse = tf.reduce_mean(tf.square(y_true - y_pred))

```

```

# Dynamic scaling based on loss magnitude
if mse < 1e-4:
    return mse * self.scale_factor
else:
    return mse

```

## Loss Function Selection Matrix for Autoencoders

| Data Type       | Recommended Loss     | Alternative    | Notes                       |
|-----------------|----------------------|----------------|-----------------------------|
| Images [0,1]    | Binary Cross-Entropy | MSE            | BCE preserves sharpness     |
| Images [0,255]  | MSE / RMSE           | Huber          | Scale appropriately         |
| Audio Signals   | MSE                  | MAE            | Consider frequency domain   |
| Tabular (mixed) | Combined MSE+BCE     | Custom         | Handle each type separately |
| Sparse Data     | MAE                  | Huber          | Robust to many zeros        |
| Time Series     | Huber                | MSE+Smoothness | Add temporal consistency    |

## Advanced Autoencoder Loss Functions

200

### 1. Perceptual Loss (for images)

```

def perceptual_loss(vgg_model, content_weight=1.0, style_weight=0.001):
    """Use pre-trained VGG features for perceptual similarity"""
    def loss(y_true, y_pred):
        # Extract features
        true_features = vgg_model(y_true)
        pred_features = vgg_model(y_pred)

        # Content loss (MSE on features)
        content_loss = tf.reduce_mean(
            tf.square(true_features - pred_features)
        )

        return content_weight * content_loss

    return loss

```

### 2. Structural Similarity (SSIM) Loss

```

def ssim_loss(y_true, y_pred, max_val=1.0):
    """Structural similarity for maintaining image structure"""

```

```

        return 1.0 - tf.reduce_mean(
            tf.image.ssim(y_true, y_pred, max_val=max_val)
    )

```

### 3. Variational Autoencoder (VAE) Loss

```

def vae_loss(z_mean, z_log_var, beta=1.0):
    """VAE loss = Reconstruction + KL divergence"""
    def loss(y_true, y_pred):
        # Reconstruction loss
        reconstruction = tf.reduce_mean(
            tf.keras.losses.binary_crossentropy(y_true, y_pred)
        ) * np.prod(y_true.shape[1:])

        # KL divergence
        kl_loss = -0.5 * tf.reduce_mean(
            1 + z_log_var - tf.square(z_mean) - tf.exp(z_log_var)
        )

        return reconstruction + beta * kl_loss

    return loss

```

## Practical Recommendations

201

1. **Start Simple:** Begin with MSE, then experiment if needed
2. **Monitor Gradients:** Use TensorBoard to track gradient magnitudes
3. **Data Preprocessing:** Proper normalization prevents extreme loss values
4. **Ensemble Approach:** Train multiple autoencoders with different losses
5. **Validation Strategy:** Use reconstruction quality metrics beyond loss

## Memory Trick: "HuBER"

**H**uber combines **B**est of both worlds, **E**ffectively **R**obust

This comprehensive guide ensures you can select and implement the most appropriate loss function for any autoencoder application, with strategies to handle the vanishing gradient problem effectively.

## **Exercise 1: Custom Loss Function**

Implement a custom loss function that:

1. Combines MSE and MAE with a learnable weight
2. Adapts the weight based on the prediction error distribution
3. Is more robust to outliers than MSE but smoother than MAE
4. Test it on a regression problem with outliers

## **Exercise 2: Network Architecture Search**

Create a system that:

1. Automatically tests different network architectures
2. Varies depth (number of layers) and width (neurons per layer)
3. Uses cross-validation to evaluate each architecture
4. Implements early stopping to save training time
5. Reports the best architecture for your dataset

## **Exercise 3: Advanced Regularization**

Implement a neural network with:

1. Custom dropout that varies by layer depth
2. L1 and L2 regularization with different strengths
3. Gradient clipping to prevent exploding gradients
4. Learning rate scheduling
5. Compare performance with and without each technique

202

## **Exercise 4: Real-time Anomaly Detection**

Build a system that:

1. Trains on normal network traffic patterns
2. Detects anomalous protocols in real-time
3. Adapts to new normal patterns over time
4. Maintains a sliding window of recent predictions
5. Triggers alerts for sustained anomalies

## **Exercise 5: Ensemble Protocol Classifier**

Create an ensemble model that:

1. Combines multiple neural networks with different architectures

2. Uses voting or averaging for final predictions
  3. Implements confidence-weighted voting
  4. Tracks individual model performance
  5. Automatically retrains underperforming models
-

This section covered the fundamentals of deep learning:

## Key Concepts Mastered:

- **Mathematical Foundations:** Understanding gradients, chain rule, and optimization
- **Activation Functions:** Non-linearity for complex patterns
- **Loss Functions:** Choosing the right objective
- **Backpropagation:** How neural networks learn
- **Regularization:** Preventing overfitting
- **Multi-class Classification:** Softmax and categorical outputs

## Critical Formulas:

| Concept          | Formula   | Memory Aid                  |
|------------------|---|-----------------------------|
| Neuron Output    | $y = \sigma(Wx + b)$  | "WIBA"                      |
| Gradient Descent | $\theta_{\text{new}} = \theta_{\text{old}} - \alpha \nabla L$ | "Step downhill"             |
| Softmax          | $e^{z_i} / \sum e^{z_j}$                                      | "Exponential normalization" |
| Cross-entropy    | $-\sum y \log(\hat{y})$                                       | "Logarithmic penalty"       |
| Dropout          | $x \times \text{Bernoulli}(p)$                                | "Random zeroing"            |

204

## Architecture Guidelines:

1. **Start Simple:** 2-3 hidden layers
2. **Hidden Layer Size:** Between input and output size
3. **Activation:** ReLU for hidden, task-specific for output
4. **Regularization:** Always use some form
5. **Monitoring:** Watch validation metrics

## Next Steps:

In Section 5, we'll explore Convolutional Neural Networks and Autoencoders, learning how to:

- Process image and text data with specialized architectures
- Build autoencoders for anomaly detection
- Create ensemble models for robust predictions
- Apply deep learning to malware detection and log analysis

# Machine Learning Training Documentation

## Section 5: Autoencoders and Convolutional Networks

### Overview

This section explores specialized neural network architectures designed for pattern recognition and anomaly detection. We'll master Convolutional Neural Networks (CNNs) for processing structured data like images and text, then dive deep into autoencoders for unsupervised anomaly detection. These architectures form the backbone of modern security systems for malware detection, log analysis, and network anomaly identification.

### Learning Objectives

By the end of this section, you will be able to:

- Understand convolution operations and their applications
- Design CNNs for image and text classification
- Master embedding layers for discrete data
- Build autoencoders for dimensionality reduction
- Implement ensemble anomaly detection systems
- Deploy real-time anomaly detection pipelines

# What is a Convolution?

A **convolution** is a mathematical operation that combines two functions to produce a third function, expressing how one function modifies the other.

Convolution in 1D:

$$(f * g)(t) = \int f(\tau)g(t-\tau)d\tau$$

Convolution in 2D (discrete):

$$(f * g)[i,j] = \sum \sum f[m,n] \times g[i-m, j-n]$$

## Visual Intuition

| Input Image   | Filter/Kernel                                  | Feature Map   |
|---|--|---|
| $\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 0 & 1 & 2 \end{bmatrix}$ | $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ | $= \begin{bmatrix} 5 & 8 & 11 \\ 17 & 20 & 23 \\ 9 & 2 & 3 \end{bmatrix}$ |

Sliding the  $2 \times 2$  kernel across the image

206

## Why Convolutions for Security Data?

| Property               | Benefit                    | Security Application                         |
|------------------------|----------------------------|--|
| Translation Invariance | Detects patterns anywhere  | Malware signatures in any file location      |
| Parameter Sharing      | Fewer parameters           | Efficient processing of large logs           |
| Local Connectivity     | Captures local patterns    | Byte sequences in executables                |
| Hierarchical Learning  | Builds complex from simple | Assembly instructions → functions → behavior |

## Types of Convolutions

| Type           | Kernel        | Use Case               | Formula                                |
|----------------|---------------|------------------------|--|
| 1D Convolution | 1D array      | Time series, sequences | $\sum x[i-k] \times w[k]$              |
| 2D Convolution | 2D matrix     | Images, spectrograms   | $\sum \sum x[i-m, j-n] \times w[m, n]$ |
| Dilated        | Sparse kernel | Large receptive field  | $\sum x[i-d \times k] \times w[k]$     |

| Type       | Kernel     | Use Case         | Formula                |
|------------|------------|------------------|------------------------|
| Transposed | Upsampling | Decoder networks | Reverse of convolution |

## Memory Trick: "SLIP"

Slide kernel, Local Interactions, Parameter sharing

---

# Core Components

Input → Conv → Activation → Pooling → Conv → ... → Flatten → Dense → Output

## Convolutional Layer Parameters

| Parameter          | Symbol | Purpose                     | Typical Values    |
|--------------------|--------|-----------------------------|-------------------|
| <b>Filters</b>     | F      | Number of feature detectors | 32, 64, 128       |
| <b>Kernel Size</b> | K×K    | Size of each filter         | 3×3, 5×5, 7×7     |
| <b>Stride</b>      | S      | Step size                   | 1 (default), 2    |
| <b>Padding</b>     | P      | Border handling             | 'same', 'valid'   |
| <b>Dilation</b>    | D      | Kernel spacing              | 1 (default), 2, 4 |

## Output Size Calculation

Output\_size =  $\lfloor (\text{Input\_size} + 2 \times \text{Padding} - \text{Kernel\_size}) / \text{Stride} \rfloor + 1$

208

Example:

Input: 32×32, Kernel: 5×5, Stride: 1, Padding: 2  
Output:  $\lfloor (32 + 2 \times 2 - 5) / 1 \rfloor + 1 = 32 \times 32$  (same)

## Pooling Operations

| Type                   | Operation          | Purpose                     | Properties             |
|------------------------|--------------------|-----------------------------|------------------------|
| <b>Max Pooling</b>     | max(region)        | Detect strongest activation | Translation invariant  |
| <b>Average Pooling</b> | mean(region)       | Smooth features             | Reduces noise          |
| <b>Global Pooling</b>  | Entire feature map | Fixed output size           | Handles variable input |

## Visual Representation of CNN Operations

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
from matplotlib.patches import Rectangle
```

```

import tensorflow as tf

# Visualize CNN operations
fig, axes = plt.subplots(2, 3, figsize=(15, 10))

# 1. Convolution Operation
ax = axes[0, 0]
ax.set_title('Convolution Operation', fontsize=14, fontweight='bold')

# Input feature map
input_size = 5
kernel_size = 3
input_map = np.random.randn(input_size, input_size)
kernel = np.array([[1, 0, -1], [2, 0, -2], [1, 0, -1]]) # Sobel filter

# Visualize
im = ax.imshow(input_map, cmap='Blues', alpha=0.8)
rect = Rectangle((0, 0), kernel_size, kernel_size,
                 linewidth=3, edgecolor='red', facecolor='none')
ax.add_patch(rect)
ax.set_xlim(-0.5, input_size-0.5)
ax.set_ylim(input_size-0.5, -0.5)
ax.text(1.5, -0.7, 'Sliding Kernel', ha='center', color='red',
        fontsize=10)

# 2. Feature Maps
ax = axes[0, 1]
ax.set_title('Multiple Feature Maps', fontsize=14, fontweight='bold')

# Simulate multiple filters
n_filters = 4
feature_maps = np.random.randn(n_filters, 8, 8)

# Stack feature maps
for i in range(n_filters):
    offset = i * 0.5
    extent = [offset, 8 + offset, offset, 8 + offset]
    ax.imshow(feature_maps[i], extent=extent, cmap='viridis', alpha=0.7)
    ax.text(4 + offset, -0.5 + offset, f'Filter {i+1}', ha='center')

ax.set_xlim(-0.5, 10)
ax.set_ylim(-1, 10)
ax.axis('off')

# 3. Pooling Operation
ax = axes[0, 2]
ax.set_title('Max Pooling (2x2)', fontsize=14, fontweight='bold')

# Create sample feature map
feature_map = np.array([

```

```

[1, 3, 2, 4],
[5, 6, 1, 2],
[3, 2, 4, 6],
[1, 4, 5, 3]
])

pooled = np.array([
    [6, 4],
    [4, 6]
])

# Show feature map with pooling regions
im = ax.imshow(feature_map, cmap='YlOrRd')
for i in range(0, 4, 2):
    for j in range(0, 4, 2):
        rect = Rectangle((j-0.5, i-0.5), 2, 2,
                          linewidth=2, edgecolor='blue', facecolor='none')
        ax.add_patch(rect)

# Add values
for i in range(4):
    for j in range(4):
        ax.text(j, i, str(feature_map[i, j]), ha='center', va='center')

ax.set_title('Input Feature Map → Pooled Output', fontsize=12) 210

# 4. Receptive Field
ax = axes[1, 0]
ax.set_title('Receptive Field Growth', fontsize=14, fontweight='bold')

layers = ['Input', 'Conv1\n(3x3)', 'Conv2\n(3x3)', 'Conv3\n(3x3)']
receptive_fields = [1, 3, 5, 7]
colors = ['blue', 'green', 'orange', 'red']

for i, (layer, rf, color) in enumerate(zip(layers, receptive_fields,
colors)):
    # Draw receptive field
    rect_size = rf / 7 * 3 # Normalize to display
    rect = Rectangle((i - rect_size/2, 1 - rect_size/2),
                     rect_size, rect_size,
                     facecolor=color, alpha=0.3, edgecolor=color,
                     linewidth=2)
    ax.add_patch(rect)
    ax.text(i, 0.2, layer, ha='center', fontsize=10)
    ax.text(i, 2.2, f'{rf}x{rf}', ha='center', fontsize=10,
fontweight='bold')

ax.set_xlim(-1, 4)
ax.set_ylim(0, 2.5)
ax.axis('off')

```

```

# 5. Padding Types
ax = axes[1, 1]
ax.set_title('Padding Strategies', fontsize=14, fontweight='bold')

# Valid padding
valid_input = np.ones((5, 5))
valid_output = np.ones((3, 3))

ax.text(0.25, 0.7, 'Valid Padding\n(No padding)', ha='center',
        fontsize=10)
ax.add_patch(Rectangle((0, 0.3), 0.5, 0.3, facecolor='lightblue',
                      edgecolor='black'))
ax.arrow(0.5, 0.45, 0.3, 0, head_width=0.05, head_length=0.05, fc='black')
ax.add_patch(Rectangle((0.9, 0.35), 0.3, 0.2, facecolor='lightcoral',
                      edgecolor='black'))

# Same padding
ax.text(0.25, 0.2, 'Same Padding\n(Preserve size)', ha='center',
        fontsize=10)
ax.add_patch(Rectangle((0, -0.2), 0.5, 0.3, facecolor='lightblue',
                      edgecolor='black'))
ax.add_patch(Rectangle((-0.05, -0.25), 0.6, 0.4, facecolor='none',
                      edgecolor='gray', linestyle='--'))
ax.arrow(0.5, -0.05, 0.3, 0, head_width=0.05, head_length=0.05,
        fc='black')
ax.add_patch(Rectangle((0.9, -0.2), 0.5, 0.3, facecolor='lightcoral',
                      edgecolor='black'))

ax.set_xlim(-0.3, 1.5)
ax.set_ylim(-0.4, 0.8)
ax.axis('off')

# 6. Architecture Visualization
ax = axes[1, 2]
ax.set_title('CNN Architecture Pattern', fontsize=14, fontweight='bold')

# Layer specifications
layers_spec = [
    ('Input\n32x32x3', 32, 3),
    ('Conv1\n32x32x32', 32, 32),
    ('Pool1\n16x16x32', 16, 32),
    ('Conv2\n16x16x64', 16, 64),
    ('Pool2\n8x8x64', 8, 64),
    ('Dense\n128', 4, 128)
]

x_positions = np.linspace(0, 1, len(layers_spec))

for i, (name, size, depth) in enumerate(layers_spec):

```

```
# Normalize sizes for visualization
height = size / 32 * 0.6
width = np.log(depth + 1) / 5 * 0.15

# Draw layer
rect = Rectangle((x_positions[i] - width/2, 0.5 - height/2),
                  width, height,
                  facecolor='skyblue' if 'Conv' in name else 'salmon',
                  edgecolor='black', linewidth=1)
ax.add_patch(rect)

# Label
ax.text(x_positions[i], 0.05, name, ha='center', fontsize=8)

# Connection
if i < len(layers_spec) - 1:
    ax.arrow(x_positions[i] + width/2, 0.5,
              x_positions[i+1] - x_positions[i] - width, 0,
              head_width=0.02, head_length=0.02, fc='gray', alpha=0.5)

ax.set_xlim(-0.1, 1.1)
ax.set_ylim(0, 1)
ax.axis('off')

plt.tight_layout()
plt.show()
```

# What are Embeddings?

Embeddings map discrete tokens (words, categories) to continuous vector representations:

```
Token ID → Dense Vector  
5      → [0.2, -0.5, 0.8, 0.1]
```

## Mathematical Definition

$$E: \mathbb{Z} \rightarrow \mathbb{R}^d$$

$$E(i) = w_i$$

Where:

- $W \in \mathbb{R}^{n \times d}$  is the embedding matrix
- $n$  = vocabulary size
- $d$  = embedding dimension

## Embedding Properties

| Property                 | Description                          | Security Application     |
|--------------------------|--------------------------------------|--------------------------|
| Dimensionality Reduction | $n$ categories → $d$ dimensions      | Reduce sparse features   |
| Semantic Similarity      | Similar items → nearby vectors       | Group similar attacks    |
| Trainable                | Learns task-specific representations | Adapt to domain          |
| Composable               | Can be combined                      | Multi-feature embeddings |

213

## Choosing Embedding Dimensions

| Vocabulary Size | Embedding Dim | Rule of Thumb                         |
|-----------------|---------------|---------------------------------------|
| < 1,000         | 8-16          | $\text{vocab\_size}^{0.25}$           |
| 1,000-10,000    | 32-64         | $\log_2(\text{vocab\_size}) \times 4$ |
| 10,000-100,000  | 128-256       | Diminishing returns                   |
| > 100,000       | 256-512       | Consider dimensionality reduction     |

## Memory Trick: "TED"



# Architecture Components

An autoencoder consists of:

$$\text{Input} \rightarrow \text{Encoder} \rightarrow \text{Latent Space} \rightarrow \text{Decoder} \rightarrow \text{Reconstruction}$$
$$X \rightarrow f(X) \rightarrow Z \rightarrow g(Z) \rightarrow \hat{X}$$

## Mathematical Formulation

$$\text{Encoder: } Z = f(X; \theta_e)$$

$$\text{Decoder: } \hat{X} = g(Z; \theta_d)$$

$$\text{Loss: } L = ||X - \hat{X}||^2 + \lambda R(\theta)$$

Where:

- $\theta_e$  = encoder parameters
- $\theta_d$  = decoder parameters
- $R(\theta)$  = regularization term
- $\lambda$  = regularization strength

## Types of Autoencoders

215

| Type              | Constraint        | Purpose           | Loss Function                        |
|-------------------|-------------------|-------------------|--------------------------------------|
| Vanilla           | None              | Basic compression | MSE                                  |
| Sparse            | L1 on activations | Feature selection | MSE + $\lambda \Sigma$               |
| Denoising         | Corrupted input   | Robustness        | MSE( $X, \hat{X}$ ) from $\tilde{X}$ |
| Variational (VAE) | Probabilistic     | Generation        | Reconstruction + KL divergence       |
| Contractive       | Jacobian penalty  | Stability         | MSE + $\lambda$                      |

## Anomaly Detection with Autoencoders

Training: Learn to reconstruct normal data

Testing: High reconstruction error = Anomaly

Threshold Selection:

- Statistical:  $\mu + k\sigma$

- Percentile: 95th, 99th
- Validation-based: Maximize F1

## Reconstruction Error Metrics

| Metric        | Formula                    | Use Case      | Sensitivity      |
|---------------|----------------------------|---------------|------------------|
| <b>MSE</b>    | $(1/n)\sum(x - \hat{x})^2$ | General       | High to outliers |
| <b>MAE</b>    | $(1/n)\sum  x - \hat{x} $  | $x - \hat{x}$ |                  |
| <b>SSIM</b>   | Structural similarity      | Images        | Perceptual       |
| <b>Cosine</b> | $1 - \cos(x, \hat{x})$     | Directional   | Angle-based      |

# Malware Detection CNN

```
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

# Generate synthetic malware image data
def create_malware_dataset(n_samples=1000):
    """Create synthetic malware vs benign executable headers"""

    # Simulate PE header visualization (32x32 grayscale)
    malware_patterns = []
    benign_patterns = []

    for _ in range(n_samples // 2):
        # Malware: specific patterns in header
        malware = np.random.rand(32, 32) * 0.5
        # Add suspicious patterns
        malware[5:10, 5:10] = 1 # Packed section
        malware[20:25, 20:25] = 0.9 # Encrypted region
        malware += np.random.randn(32, 32) * 0.1
        malware_patterns.append(malware)

        # Benign: normal structure
        benign = np.random.rand(32, 32) * 0.3
        benign[0:5, :] = 0.7 # Standard header
        benign[15:20, 10:20] = 0.6 # Normal sections
        benign += np.random.randn(32, 32) * 0.1
        benign_patterns.append(benign)

    X = np.array(malware_patterns + benign_patterns)
    y = np.array([1] * len(malware_patterns) + [0] * len(benign_patterns))

    # Shuffle
    indices = np.random.permutation(len(X))
    return X[indices], y[indices]

# Create dataset
X_malware, y_malware = create_malware_dataset(2000)
X_malware = X_malware.reshape(-1, 32, 32, 1) # Add channel dimension

# Split data
X_train, X_test, y_train, y_test = train_test_split(
    X_malware, y_malware, test_size=0.2, random_state=42
)
```

```

# Build CNN architecture with explanations
def build_malware_cnn():
    """CNN for malware detection with architectural choices explained"""

    model = keras.Sequential([
        # First conv block - detect basic patterns
        layers.Conv2D(32, (3, 3), activation='relu', padding='same',
                     input_shape=(32, 32, 1), name='feature_detector_1'),
        layers.BatchNormalization(),
        layers.MaxPooling2D((2, 2), name='downsample_1'),

        # Second conv block - combine patterns
        layers.Conv2D(64, (3, 3), activation='relu', padding='same',
                     name='feature_combiner_1'),
        layers.BatchNormalization(),
        layers.Conv2D(64, (3, 3), activation='relu', padding='same',
                     name='feature_combiner_2'),
        layers.MaxPooling2D((2, 2), name='downsample_2'),

        # Third conv block - high-level features
        layers.Conv2D(128, (3, 3), activation='relu', padding='same',
                     name='abstract_features'),
        layers.GlobalAveragePooling2D(name='global_features'),
        # Classification head
        layers.Dense(64, activation='relu', name='classifier_hidden'),
        layers.Dropout(0.5),
        layers.Dense(1, activation='sigmoid', name='malware_probability')
    ])

    return model

# Build and compile model
malware_cnn = build_malware_cnn()
malware_cnn.compile(
    optimizer='adam',
    loss='binary_crossentropy',
    metrics=['accuracy', tf.keras.metrics.AUC(name='auc')])
)

# Visualize model architecture
print(malware_cnn.summary())

# Train model
history = malware_cnn.fit(
    X_train, y_train,
    validation_split=0.2,
    epochs=20,
    batch_size=32,
)

```

```

    verbose=0
)

# Visualize feature maps
def visualize_cnn_features(model, sample_image):
    """Visualize what CNN layers learn"""

    # Create feature extractor for each conv layer
    layer_outputs = []
    layer_names = []

    for layer in model.layers:
        if 'conv' in layer.name:
            layer_outputs.append(layer.output)
            layer_names.append(layer.name)

    feature_model = keras.Model(inputs=model.input, outputs=layer_outputs)
    features = feature_model.predict(sample_image[np.newaxis, ...])

    # Plot features
    fig, axes = plt.subplots(len(features), 8, figsize=(16,
2*len(features)))

    for layer_idx, (feature_map, layer_name) in enumerate(zip(features,
layer_names)):
        for filter_idx in range(min(8, feature_map.shape[-1])):
            ax = axes[layer_idx, filter_idx] if len(features) > 1 else
axes[filter_idx]
            ax.imshow(feature_map[0, :, :, filter_idx], cmap='viridis')
            ax.axis('off')
            if filter_idx == 0:
                ax.set_title(f'{layer_name}', fontsize=10)

    plt.suptitle('CNN Feature Maps - What Each Layer Sees', fontsize=14)
    plt.tight_layout()
    plt.show()

# Visualize learned features
test_malware = X_test[y_test == 1][0]
test_benign = X_test[y_test == 0][0]

print("Malware Sample Features:")
visualize_cnn_features(malware_cnn, test_malware)

print("\nBenign Sample Features:")
visualize_cnn_features(malware_cnn, test_benign)

```

```

# CNN for text classification (e.g., spam detection)
def build_text_cnn(vocab_size, embedding_dim, max_length):
    """1D CNN for text classification"""

    model = keras.Sequential([
        # Embedding layer
        layers.Embedding(vocab_size, embedding_dim,
                          input_length=max_length,
                          name='word_embeddings'),

        # Multiple filter sizes (like n-grams)
        # This would be better with Functional API, but showing sequential
        layers.Conv1D(128, 3, activation='relu', padding='same',
                      name='trigram_detector'),
        layers.GlobalMaxPooling1D(name='important_trigrams'),

        # Classification
        layers.Dense(64, activation='relu'),
        layers.Dropout(0.5),
        layers.Dense(1, activation='sigmoid', name='spam_probability')
    ])

    return model

```

220

```

# Visualize text CNN concept
fig, axes = plt.subplots(1, 3, figsize=(15, 5))

# 1. Embedding visualization
ax = axes[0]
ax.set_title('Word Embeddings', fontsize=14, fontweight='bold')

words = ['click', 'free', 'money', 'hello', 'meeting']
embeddings = np.random.randn(5, 3) # 3D embeddings for visualization

ax.scatter(embeddings[:3, 0], embeddings[:3, 1], c='red', s=100,
           label='Spam words')
ax.scatter(embeddings[3:, 0], embeddings[3:, 1], c='blue', s=100,
           label='Normal words')

for i, word in enumerate(words):
    ax.annotate(word, (embeddings[i, 0], embeddings[i, 1]),
                xytext=(5, 5), textcoords='offset points')

ax.set_xlabel('Embedding Dim 1')
ax.set_ylabel('Embedding Dim 2')
ax.legend()
ax.grid(True, alpha=0.3)

```

```

# 2. 1D Convolution on text
ax = axes[1]
ax.set_title('1D Convolution on Text', fontsize=14, fontweight='bold')

# Simulate embedded sentence
sentence = "Click here for free money now"
embedded = np.random.randn(6, 8) # 6 words, 8-dim embeddings

im = ax.imshow(embedded.T, aspect='auto', cmap='coolwarm')
ax.set_yticks(range(8))
ax.set_ylabel('Embedding Dimensions')
ax.set_xticks(range(6))
ax.set_xticklabels(sentence.split(), rotation=45)

# Show convolution window
from matplotlib.patches import Rectangle
rect = Rectangle((0.5, -0.5), 3, 8, linewidth=3,
                 edgecolor='green', facecolor='none')
ax.add_patch(rect)
ax.text(2, -1.5, '3-word filter', ha='center', color='green')

# 3. Feature extraction
ax = axes[2]
ax.set_title('Max Pooling Results', fontsize=14, fontweight='bold')

filter_responses = np.random.rand(4, 10) # 4 positions, 10 filters
max_responses = filter_responses.max(axis=0)

bars = ax.bar(range(10), max_responses)
for i in range(3):
    bars[i].set_color('red')
ax.set_xlabel('Filter Index')
ax.set_ylabel('Max Activation')
ax.text(1, 0.9, 'Spam-related\nfilters', ha='center', color='red')

plt.tight_layout()
plt.show()

```

# Building Robust Autoencoders

```
# Create comprehensive autoencoder implementations
class AutoencoderFactory:
    """Factory for different autoencoder types"""

    @staticmethod
    def create_vanilla_autoencoder(input_dim, encoding_dim):
        """Standard autoencoder"""
        # Encoder
        encoder_input = layers.Input(shape=(input_dim,))
        encoded = layers.Dense(128, activation='relu')(encoder_input)
        encoded = layers.Dense(64, activation='relu')(encoded)
        encoded = layers.Dense(encoding_dim, activation='relu',
                               name='latent_space')(encoded)

        # Decoder
        decoded = layers.Dense(64, activation='relu')(encoded)
        decoded = layers.Dense(128, activation='relu')(decoded)
        decoded = layers.Dense(input_dim, activation='sigmoid')(decoded)

        # Models
        autoencoder = keras.Model(encoder_input, decoded)
        encoder = keras.Model(encoder_input, encoded)

        return autoencoder, encoder

    @staticmethod
    def create_sparse_autoencoder(input_dim, encoding_dim, sparsity=0.05):
        """Sparse autoencoder with L1 regularization"""
        from tensorflow.keras import regularizers

        encoder_input = layers.Input(shape=(input_dim,))
        encoded = layers.Dense(128, activation='relu')(encoder_input)
        encoded = layers.Dense(
            encoding_dim,
            activation='relu',
            activity_regularizer=regularizers.l1(sparsity),
            name='sparse_latent'
        )(encoded)

        decoded = layers.Dense(128, activation='relu')(encoded)
        decoded = layers.Dense(input_dim, activation='sigmoid')(decoded)

        autoencoder = keras.Model(encoder_input, decoded)
        encoder = keras.Model(encoder_input, encoded)

        return autoencoder, encoder
```

```

    @staticmethod
    def create_denoising_autoencoder(input_dim, encoding_dim,
noise_factor=0.1):
        """Denoising autoencoder"""
        # Clean input for encoder output
        clean_input = layers.Input(shape=(input_dim,))

        # Noisy input for training
        noisy_input = layers.Input(shape=(input_dim,))

        # Shared encoder
        encoder_layers = keras.Sequential([
            layers.Dense(128, activation='relu'),
            layers.Dense(64, activation='relu'),
            layers.Dense(encoding_dim, activation='relu')
        ])

        # Shared decoder
        decoder_layers = keras.Sequential([
            layers.Dense(64, activation='relu'),
            layers.Dense(128, activation='relu'),
            layers.Dense(input_dim, activation='sigmoid')
        ])

        # Build models
        encoded_clean = encoder_layers(clean_input)
        encoded_noisy = encoder_layers(noisy_input)
        decoded = decoder_layers(encoded_noisy)

        autoencoder = keras.Model([noisy_input, clean_input], decoded)
        encoder = keras.Model(clean_input, encoded_clean)

        return autoencoder, encoder

# Visualize autoencoder architectures
def visualize_autoencoder_architectures():
    """Compare different autoencoder types"""

    fig, axes = plt.subplots(2, 2, figsize=(12, 10))

    # 1. Vanilla Autoencoder
    ax = axes[0, 0]
    ax.set_title('Vanilla Autoencoder', fontsize=14, fontweight='bold')

    layers_vanilla = [100, 128, 64, 32, 64, 128, 100]
    layer_names = ['Input', 'Hidden1', 'Hidden2', 'Latent', 'Hidden3',
'Hidden4', 'Output']

    for i, (size, name) in enumerate(zip(layers_vanilla, layer_names)):

```

```

y_positions = np.linspace(0.2, 0.8, min(size, 10))
x_position = i / (len(layers_vanilla) - 1)

# Draw neurons
ax.scatter([x_position] * len(y_positions), y_positions,
           s=50, c='blue' if i < 4 else 'green', alpha=0.6)
ax.text(x_position, 0.05, name, ha='center', fontsize=8)

# Draw connections
if i < len(layers_vanilla) - 1:
    next_y = np.linspace(0.2, 0.8, min(layers_vanilla[i+1], 10))
    for y1 in y_positions[::3]: # Sample connections
        for y2 in next_y[::3]:
            ax.plot([x_position, (i+1)/(len(layers_vanilla)-1)],
                    [y1, y2], 'gray', alpha=0.1, linewidth=0.5)

ax.text(0.5, 0.95, 'Reconstruction Loss: ||X -  $\hat{X}$ ||2', ha='center',
        fontsize=10)
ax.set_xlim(-0.1, 1.1)
ax.set_ylim(0, 1)
ax.axis('off')

# 2. Sparse Autoencoder
ax = axes[0, 1]
ax.set_title('Sparse Autoencoder', fontsize=14, fontweight='bold') 224

# Visualize sparsity constraint
latent_activations = np.random.exponential(0.1, 32)
latent_activations[latent_activations > 0.5] = 0 # Sparsity

bars = ax.bar(range(32), latent_activations)
active_neurons = np.where(latent_activations > 0.1)[0]
for idx in active_neurons:
    bars[idx].set_color('red')

ax.set_xlabel('Latent Neuron Index')
ax.set_ylabel('Activation Level')
ax.text(16, 0.4, 'Sparse Activations\n(Most neurons inactive)',
        ha='center', bbox=dict(boxstyle="round,pad=0.3",
        facecolor="yellow"))
ax.text(16, 0.5, 'Loss = MSE +  $\lambda||h||^1$ ', ha='center', fontsize=10)

# 3. Denoising Autoencoder
ax = axes[1, 0]
ax.set_title('Denoising Autoencoder', fontsize=14, fontweight='bold')

# Show denoising process
original = np.sin(np.linspace(0, 4*np.pi, 100))
noise = np.random.normal(0, 0.3, 100)
noisy = original + noise

```

```

reconstructed = original + np.random.normal(0, 0.05, 100) # Small
residual error

ax.plot(original, 'g-', label='Original', linewidth=2)
ax.plot(noisy, 'r.', label='Noisy Input', alpha=0.5, markersize=3)
ax.plot(reconstructed, 'b--', label='Reconstructed', linewidth=2)
ax.legend()
ax.set_xlabel('Time')
ax.set_ylabel('Signal')
ax.text(50, -1.5, 'Learns to remove noise', ha='center',
       bbox=dict(boxstyle="round, pad=0.3", facecolor="lightblue"))

# 4. Variational Autoencoder
ax = axes[1, 1]
ax.set_title('Variational Autoencoder (VAE)', fontsize=14,
fontweight='bold')

# Visualize latent space distribution
from scipy.stats import multivariate_normal

x = np.linspace(-3, 3, 100)
y = np.linspace(-3, 3, 100)
X, Y = np.meshgrid(x, y)
pos = np.dstack((X, Y))

# Two different classes in latent space
rv1 = multivariate_normal([-1, -1], [[0.5, 0], [0, 0.5]])
rv2 = multivariate_normal([1, 1], [[0.5, 0], [0, 0.5]])

ax.contour(X, Y, rv1.pdf(pos), colors='blue', alpha=0.5)
ax.contour(X, Y, rv2.pdf(pos), colors='red', alpha=0.5)

# Sample points
samples1 = np.random.multivariate_normal([-1, -1], [[0.5, 0], [0, 0.5]], 20)
samples2 = np.random.multivariate_normal([1, 1], [[0.5, 0], [0, 0.5]], 20)

ax.scatter(samples1[:, 0], samples1[:, 1], c='blue', alpha=0.6,
label='Class 1')
ax.scatter(samples2[:, 0], samples2[:, 1], c='red', alpha=0.6,
label='Class 2')

ax.set_xlabel('Latent Dimension 1')
ax.set_ylabel('Latent Dimension 2')
ax.legend()
ax.text(0, -3.5, 'Loss = Reconstruction + KL(q(z|x) || p(z))',
       ha='center', fontsize=10)

plt.tight_layout()

```

```
plt.show()  
visualize_autoencoder_architectures()
```

---

# Why Ensemble Autoencoders?

| Single Model Issues          | Ensemble Solution                         |
|------------------------------|---|
| High false positives         | Voting reduces noise                      |
| Misses certain anomalies     | Different models catch different patterns |
| Sensitive to hyperparameters | Robustness through diversity              |
| Fixed threshold              | Dynamic thresholding                      |

## Ensemble Strategies

```
# Ensemble anomaly detection system
class EnsembleAnomalyDetector:
    """Combine multiple autoencoders for robust anomaly detection"""

    def __init__(self, input_dim):
        self.input_dim = input_dim
        self.models = []
        self.thresholds = []

    def build_ensemble(self):
        """Create diverse autoencoder ensemble"""

        # Model 1: Shallow autoencoder
        ae1, enc1 = AutoencoderFactory.create_vanilla_autoencoder(
            self.input_dim, encoding_dim=16
        )

        # Model 2: Deep autoencoder
        ae2, enc2 = AutoencoderFactory.create_vanilla_autoencoder(
            self.input_dim, encoding_dim=8
        )

        # Model 3: Sparse autoencoder
        ae3, enc3 = AutoencoderFactory.create_sparse_autoencoder(
            self.input_dim, encoding_dim=32, sparsity=0.1
        )

        self.models = [
            ('shallow', ae1, enc1),
            ('deep', ae2, enc2),
            ('sparse', ae3, enc3)
        ]

        # Compile all models
```

```

        for name, ae, _ in self.models:
            ae.compile(optimizer='adam', loss='mse')

    def train_ensemble(self, X_normal, validation_split=0.2):
        """Train all models and set thresholds"""

        histories = {}

        for name, ae, _ in self.models:
            print(f"Training {name} autoencoder...")
            history = ae.fit(
                X_normal, X_normal,
                epochs=50,
                batch_size=32,
                validation_split=validation_split,
                verbose=0
            )
            histories[name] = history

            # Calculate threshold based on validation data
            val_pred = ae.predict(X_normal[-int(len(X_normal)*validation_split):])
            val_errors = np.mean((X_normal[-int(len(X_normal)*validation_split):] - val_pred)**2, axis=1)
            threshold = np.percentile(val_errors, 95)
            self.thresholds.append(threshold)

        return histories

    def detect_anomalies(self, X_test, strategy='voting'):
        """Detect anomalies using ensemble"""

        predictions = []
        scores = []

        for (name, ae, _), threshold in zip(self.models, self.thresholds):
            # Get reconstruction error
            X_pred = ae.predict(X_test)
            errors = np.mean((X_test - X_pred)**2, axis=1)

            # Binary predictions
            preds = (errors > threshold).astype(int)
            predictions.append(preds)

            # Normalized scores
            scores.append(errors / threshold)

        predictions = np.array(predictions)
        scores = np.array(scores)

```

```
if strategy == 'voting':
    # Majority voting
    final_predictions = (predictions.sum(axis=0) >= 2).astype(int)
elif strategy == 'average':
    # Average score
    final_scores = scores.mean(axis=0)
    final_predictions = (final_scores > 1).astype(int)
elif strategy == 'max':
    # Most suspicious score
    final_scores = scores.max(axis=0)
    final_predictions = (final_scores > 1).astype(int)
else:
    raise ValueError(f"Unknown strategy: {strategy}")

return final_predictions, scores

def visualize_ensemble_performance(self, X_test, y_test):
    """Visualize how ensemble improves detection"""

    # Get individual model predictions
    individual_predictions = []
    individual_scores = []

    for (name, ae, _), threshold in zip(self.models, self.thresholds):
        X_pred = ae.predict(X_test)
        errors = np.mean((X_test - X_pred)**2, axis=1)
        preds = (errors > threshold).astype(int)
        individual_predictions.append(preds)
        individual_scores.append(errors)

    # Get ensemble predictions
    ensemble_preds, _ = self.detect_anomalies(X_test,
                                                strategy='voting')

    # Calculate metrics
    from sklearn.metrics import precision_score, recall_score,
f1_score

    results = []
    for i, (name, _, _) in enumerate(self.models):
        precision = precision_score(y_test, individual_predictions[i])
        recall = recall_score(y_test, individual_predictions[i])
        f1 = f1_score(y_test, individual_predictions[i])
        results.append((name, precision, recall, f1))

    # Ensemble results
    ens_precision = precision_score(y_test, ensemble_preds)
    ens_recall = recall_score(y_test, ensemble_preds)
    ens_f1 = f1_score(y_test, ensemble_preds)
    results.append(('Ensemble', ens_precision, ens_recall, ens_f1))
```

```

# Visualize
fig, axes = plt.subplots(1, 2, figsize=(12, 5))

# Bar chart of metrics
ax = axes[0]
models = [r[0] for r in results]
metrics = np.array([[r[1], r[2], r[3]] for r in results])

x = np.arange(len(models))
width = 0.25

    ax.bar(x - width, metrics[:, 0], width, label='Precision',
color='blue')
    ax.bar(x, metrics[:, 1], width, label='Recall', color='green')
    ax.bar(x + width, metrics[:, 2], width, label='F1-Score',
color='red')

ax.set_xlabel('Model')
ax.set_ylabel('Score')
ax.set_title('Individual vs Ensemble Performance')
ax.set_xticks(x)
ax.set_xticklabels(models, rotation=45)
ax.legend()
ax.grid(True, alpha=0.3)

# Venn diagram showing agreement
ax = axes[1]
from matplotlib.patches import Circle

# Draw three circles
circle1 = Circle((0.35, 0.5), 0.3, alpha=0.3, color='blue')
circle2 = Circle((0.65, 0.5), 0.3, alpha=0.3, color='green')
circle3 = Circle((0.5, 0.3), 0.3, alpha=0.3, color='red')

ax.add_patch(circle1)
ax.add_patch(circle2)
ax.add_patch(circle3)

# Calculate overlaps
all_agree = np.sum((individual_predictions[0] ==
individual_predictions[1]) &
                    (individual_predictions[1] ==
individual_predictions[2]))

ax.text(0.5, 0.5, f'{all_agree}\nagree', ha='center', fontsize=10)
ax.text(0.2, 0.7, self.models[0][0], fontsize=10)
ax.text(0.8, 0.7, self.models[1][0], fontsize=10)
ax.text(0.5, 0.1, self.models[2][0], fontsize=10)

```

```

        ax.set_xlim(0, 1)
        ax.set_ylim(0, 1)
        ax.set_aspect('equal')
        ax.axis('off')
        ax.set_title('Model Agreement on Anomalies')

    plt.tight_layout()
    plt.show()

# Demonstrate ensemble system
# Create anomaly detection dataset
from sklearn.datasets import make_classification

X_ensemble, y_ensemble = make_classification(
    n_samples=1000,
    n_features=20,
    n_informative=15,
    n_redundant=5,
    n_classes=2,
    weights=[0.9, 0.1], # 10% anomalies
    random_state=42
)

# Use only normal data for training
X_train_normal = X_ensemble[y_ensemble == 0][:500]
X_test_ensemble = X_ensemble[500:]
y_test_ensemble = y_ensemble[500:]

# Build and train ensemble
ensemble_detector = EnsembleAnomalyDetector(input_dim=20)
ensemble_detector.build_ensemble()
ensemble_detector.train_ensemble(X_train_normal)

# Evaluate
ensemble_detector.visualize_ensemble_performance(X_test_ensemble,
y_test_ensemble)

```

231

## 5.8 Advanced Reconstruction Metrics

### Beyond Basic Loss Functions

While MSE is common, sophisticated anomaly detection requires multiple metrics:

### Comprehensive Reconstruction Metrics

| Metric                              | Formula                                       | Use Case                 | Advantages                                   |
|-------------------------------------|---|--------------------------|--|
| <b>Per-Feature MSE</b>              | $(1/n)\sum(x_i - \hat{x}_i)^2$ per feature    | Feature importance       | Identifies which features reconstruct poorly |
| <b>Normalized MSE</b>               | MSE / $\text{var}(X)$                         | Cross-dataset comparison | Scale-independent                            |
| <b>Peak Signal-to-Noise Ratio</b>   | $10 \cdot \log_{10}(\text{MAX}^2/\text{MSE})$ | Image quality            | Standard in image processing                 |
| <b>Structural Similarity (SSIM)</b> | $f(\text{luminance, contrast, structure})$    | Perceptual similarity    | Better matches human perception              |
| <b>Cosine Similarity</b>            | $X \cdot \hat{X} / (\ X\ \ \hat{X}\ )$        | Direction vs magnitude   | Good for high-dimensional data               |
| <b>KL Divergence</b>                | $\sum X \cdot \log(X/\hat{X})$                | Distribution matching    | For probabilistic data                       |

## Implementation of Advanced Metrics

```

import numpy as np
import tensorflow as tf
from skimage.metrics import structural_similarity

class AdvancedReconstructionMetrics:
    """
    Comprehensive reconstruction metrics for autoencoders
    """

    @staticmethod
    def per_feature_mse(original, reconstructed):
        """Calculate MSE for each feature separately"""
        return np.mean((original - reconstructed)**2, axis=0)

    @staticmethod
    def normalized_mse(original, reconstructed):
        """MSE normalized by variance"""
        mse = np.mean((original - reconstructed)**2)
        variance = np.var(original)
        return mse / (variance + 1e-8)

    @staticmethod
    def psnr(original, reconstructed, max_val=1.0):
        """Peak Signal-to-Noise Ratio"""
        mse = np.mean((original - reconstructed)**2)
        if mse == 0:
            return float('inf')
        return 20 * np.log10(max_val) - 10 * np.log10(mse)

```

```
@staticmethod
def ssim_metric(original, reconstructed):
    """Structural Similarity Index"""
    # For batch processing
    ssim_values = []
    for i in range(len(original)):
        ssim_val = structural_similarity(
            original[i], reconstructed[i],
            multichannel=True if len(original[i].shape) > 2 else False
        )
        ssim_values.append(ssim_val)
    return np.mean(ssim_values)

@staticmethod
def cosine_similarity(original, reconstructed):
    """Cosine similarity per sample"""
    dot_product = np.sum(original * reconstructed, axis=1)
    norm_original = np.linalg.norm(original, axis=1)
    norm_reconstructed = np.linalg.norm(reconstructed, axis=1)
    return dot_product / (norm_original * norm_reconstructed + 1e-8)

@staticmethod
def reconstruction_probability(errors, contamination=0.1):
    """Convert errors to anomaly probabilities"""
    threshold = np.percentile(errors, 100 * (1 - contamination))
    probabilities = 1 - np.exp(-errors / threshold)
    return probabilities

# Composite anomaly score
class CompositeAnomalyScorer:
    """
    Combines multiple metrics for robust anomaly detection
    """
    def __init__(self, weights=None):
        self.weights = weights or {
            'mse': 0.3,
            'cosine': 0.2,
            'psnr': 0.2,
            'per_feature': 0.3
        }
        self.scaler = StandardScaler()
        self.is_fitted = False

    def fit(self, normal_data, autoencoder):
        """Fit on normal data to learn thresholds"""
        reconstructed = autoencoder.predict(normal_data)

        # Calculate all metrics
        self.metrics_ = {
```

```
'mse': np.mean((normal_data - reconstructed)**2, axis=1),
    'cosine': 1 - self._cosine_similarity(normal_data,
reconstructed),
    'psnr': self._batch_psnr(normal_data, reconstructed),
    'per_feature': np.max(self._per_feature_mse(normal_data,
reconstructed), axis=1)
}

# Fit scaler on normal data metrics
metric_matrix = np.column_stack(list(self.metrics_.values()))
self.scaler.fit(metric_matrix)
self.is_fitted = True

return self

def score(self, data, autoencoder):
    """Calculate composite anomaly score"""
    if not self.is_fitted:
        raise ValueError("Must fit on normal data first")

    reconstructed = autoencoder.predict(data)

    # Calculate metrics
    metrics = {
        'mse': np.mean((data - reconstructed)**2, axis=1),
        'cosine': 1 - self._cosine_similarity(data, reconstructed),
        'psnr': self._batch_psnr(data, reconstructed),
        'per_feature': np.max(self._per_feature_mse(data,
reconstructed), axis=1)
    }

    # Normalize and combine
    metric_matrix = np.column_stack(list(metrics.values()))
    normalized = self.scaler.transform(metric_matrix)

    # Weighted sum
    weights_array = np.array(list(self.weights.values()))
    composite_score = np.dot(normalized, weights_array)

    return composite_score, metrics
```

# Motivation for Ensembles

Single autoencoders can be sensitive to:

- Architecture choices
- Initialization
- Training data sampling
- Hyperparameters

Ensembles provide:

- **Robustness**: Reduced false positives
- **Coverage**: Different models catch different anomalies
- **Confidence**: Agreement indicates reliability

## Ensemble Architectures

```
class EnsembleAutoencoder:  
    """  
        Ensemble of diverse autoencoders for robust anomaly detection  
    """  
  
    def __init__(self, n_models=5):  
        self.n_models = n_models  
        self.models = []  
        self.architectures = [  
            # Diverse architectures  
            {'encoding_dim': 32, 'hidden_layers': [128, 64]},  
            {'encoding_dim': 16, 'hidden_layers': [64, 32]},  
            {'encoding_dim': 48, 'hidden_layers': [256, 128]},  
            {'encoding_dim': 24, 'hidden_layers': [128, 64, 32]},  
            {'encoding_dim': 32, 'hidden_layers': [100, 50]},  
        ]  
  
    def build_models(self, input_dim):  
        """Create diverse autoencoder architectures"""  
  
        for i in range(self.n_models):  
            arch = self.architectures[i]  
  
            # Encoder  
            encoder_input = layers.Input(shape=(input_dim,))  
            encoded = encoder_input  
  
            for units in arch['hidden_layers']:  
                encoded = layers.Dense(units, activation='relu')(encoded)  
                encoded = layers.Dropout(0.2)(encoded)
```

```

        encoded = layers.Dense(arch['encoding_dim'],
activation='relu')(encoded)

# Decoder
decoded = encoded
for units in reversed(arch['hidden_layers']):
    decoded = layers.Dense(units, activation='relu')(decoded)

decoded = layers.Dense(input_dim, activation='sigmoid')
(decoded)

# Model
autoencoder = keras.Model(encoder_input, decoded)
autoencoder.compile(optimizer='adam', loss='mse')

self.models.append(autoencoder)

def fit(self, X_train, validation_split=0.1, epochs=50):
    """Train all models with different strategies"""
    histories = []

    for i, model in enumerate(self.models):
        print(f"Training model {i+1}/{self.n_models}")

        # Different training strategies
        if i == 0:
            # Standard training
            history = model.fit(
                X_train, X_train,
                epochs=epochs,
                batch_size=32,
                validation_split=validation_split,
                verbose=0
            )
        elif i == 1:
            # With noise injection
            noise_factor = 0.1
            X_noisy = X_train + noise_factor * np.random.normal(0, 1,
X_train.shape)
            X_noisy = np.clip(X_noisy, 0., 1.)

            history = model.fit(
                X_noisy, X_train,
                epochs=epochs,
                batch_size=32,
                validation_split=validation_split,
                verbose=0
            )
        elif i == 2:

```

```

        # With dropout in input
        history = model.fit(
            X_train, X_train,
            epochs=epochs,
            batch_size=64, # Different batch size
            validation_split=validation_split,
            verbose=0
        )
    elif i == 3:
        # With L2 regularization
        for layer in model.layers:
            if hasattr(layer, 'kernel_regularizer'):
                layer.kernel_regularizer = regularizers.l2(0.01)

        model.compile(optimizer='adam', loss='mse')
        history = model.fit(
            X_train, X_train,
            epochs=epochs,
            batch_size=32,
            validation_split=validation_split,
            verbose=0
        )
    else:
        # With different optimizer
        model.compile(optimizer='rmsprop', loss='mse')
        history = model.fit(
            X_train, X_train,
            epochs=epochs,
            batch_size=48,
            validation_split=validation_split,
            verbose=0
        )
    histories.append(history)

return histories

def predict_ensemble(self, X, strategy='mean'):
    """Ensemble prediction strategies"""
    reconstructions = []
    reconstruction_errors = []

    for model in self.models:
        recon = model.predict(X)
        reconstructions.append(recon)
        errors = np.mean((X - recon)**2, axis=1)
        reconstruction_errors.append(errors)

    reconstructions = np.array(reconstructions)
    reconstruction_errors = np.array(reconstruction_errors)

```

```

if strategy == 'mean':
    # Average reconstruction
    ensemble_reconstruction = np.mean(reconstructions, axis=0)
    ensemble_error = np.mean((X - ensemble_reconstruction)**2,
axis=1)

elif strategy == 'median':
    # Median reconstruction
    ensemble_reconstruction = np.median(reconstructions, axis=0)
    ensemble_error = np.mean((X - ensemble_reconstruction)**2,
axis=1)

elif strategy == 'min':
    # Best reconstruction
    min_error_idx = np.argmin(reconstruction_errors, axis=0)
    ensemble_reconstruction = reconstructions[min_error_idx,
np.arange(len(X))]
    ensemble_error = np.min(reconstruction_errors, axis=0)

elif strategy == 'vote':
    # Voting based on threshold
    threshold = np.percentile(reconstruction_errors, 95, axis=1)
    votes = reconstruction_errors > threshold[:, np.newaxis]
    anomaly_votes = np.sum(votes, axis=0)
    ensemble_error = anomaly_votes / self.n_models
    ensemble_reconstruction = np.mean(reconstructions, axis=0)

    238

elif strategy == 'weighted':
    # Weight by inverse error
    weights = 1.0 / (reconstruction_errors + 1e-8)
    weights = weights / np.sum(weights, axis=0)
    ensemble_reconstruction = np.sum(reconstructions * weights[:, :, np.newaxis], axis=0)
    ensemble_error = np.mean((X - ensemble_reconstruction)**2,
axis=1)

return ensemble_reconstruction, ensemble_error

def detect_anomalies(self, X, contamination=0.1, strategy='vote'):
    """Detect anomalies using ensemble"""
    _, ensemble_errors = self.predict_ensemble(X, strategy=strategy)

    if strategy == 'vote':
        # For voting, higher score = more anomalous
        threshold = contamination * self.n_models
        anomalies = ensemble_errors > threshold
    else:
        # For error-based, use percentile
        threshold = np.percentile(ensemble_errors, 100 * (1 -

```

```

contamination))
    anomalies = ensemble_errors > threshold

    return anomalies, ensemble_errors

# Advanced ensemble with different model types
class HybridEnsembleDetector:
    """
    Combines different types of models for comprehensive detection
    """

    def __init__(self):
        self.models = {
            'vanilla_ae': None,
            'sparse_ae': None,
            'denoising_ae': None,
            'vae': None,
            'isolation_forest': IsolationForest(contamination=0.1),
            'one_class_svm': OneClassSVM(nu=0.1)
        }

    def build_autoencoders(self, input_dim):
        """Build different autoencoder types"""
        # Implementation of each autoencoder type
        # ... (use implementations from previous sections)
        pass

    def fit(self, X_normal):
        """Fit all models"""
        # Fit each model appropriately
        for name, model in self.models.items():
            if 'ae' in name:
                # Autoencoder training
                model.fit(X_normal, X_normal, epochs=50, verbose=0)
            else:
                # Traditional anomaly detectors
                model.fit(X_normal)

    def predict(self, X):
        """Combine predictions from all models"""
        predictions = {}

        for name, model in self.models.items():
            if 'ae' in name:
                recon = model.predict(X)
                error = np.mean((X - recon)**2, axis=1)
                predictions[name] = error
            else:
                # -1 for anomaly, 1 for normal
                pred = model.predict(X)

```

```
predictions[name] = (pred == -1).astype(float)
```

```
return predictions
```

## Visualization of Ensemble Performance

```
def visualize_ensemble_performance(X_test, y_test, ensemble_detector):
    """Visualize how ensemble improves detection"""

    # Get predictions from all models
    predictions = ensemble_detector.predict(X_test)

    fig, axes = plt.subplots(2, 3, figsize=(15, 10))
    axes = axes.ravel()

    # Plot ROC curves for each model
    from sklearn.metrics import roc_curve, auc

    for idx, (name, scores) in enumerate(predictions.items()):
        if idx < 6:
            fpr, tpr, _ = roc_curve(y_test, scores)
            roc_auc = auc(fpr, tpr)

            axes[idx].plot(fpr, tpr, 'b-', linewidth=2,
                           label=f'{name} (AUC = {roc_auc:.3f})')
            axes[idx].plot([0, 1], [0, 1], 'r--', linewidth=1)
            axes[idx].set_xlabel('False Positive Rate')
            axes[idx].set_ylabel('True Positive Rate')
            axes[idx].set_title(f'{name} ROC Curve')
            axes[idx].legend()
            axes[idx].grid(True, alpha=0.3)

    plt.tight_layout()
    plt.show()

    # Ensemble combination visualization
    fig, ax = plt.subplots(figsize=(10, 6))

    # Convert to DataFrame for easier handling
    import pandas as pd
    scores_df = pd.DataFrame(predictions)

    # Calculate ensemble score (mean of all methods)
    ensemble_score = scores_df.mean(axis=1)

    # Plot individual vs ensemble
    for method in scores_df.columns[:3]: # First 3 methods
        ax.scatter(scores_df[method], ensemble_score, alpha=0.5,
```

```
label=method)

ax.plot([0, 1], [0, 1], 'k--', linewidth=1)
ax.set_xlabel('Individual Model Score')
ax.set_ylabel('Ensemble Score')
ax.set_title('Individual vs Ensemble Anomaly Scores')
ax.legend()
ax.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()
```

## Memory Trick: "DIVERSE"

Different architectures, Independent training, Voting strategies, Error aggregation, Robust detection, Scalable deployment, Evaluation metrics

# Production Deployment Architecture

Network Traffic → Feature Extraction → Ensemble Autoencoders → Alert System



Feature Cache



Model Updates

## Implementation Best Practices

| Component          | Best Practice       | Reason                    |
|--------------------|---------------------|---------------------------|
| Feature Extraction | Sliding windows     | Capture temporal patterns |
| Normalization      | Online statistics   | Handle drift              |
| Model Updates      | Periodic retraining | Adapt to new patterns     |
| Thresholding       | Dynamic percentiles | Reduce false positives    |
| Alert Fatigue      | Rate limiting       | Maintainable operations   |

## **Exercise 1: Multi-Scale CNN**

Build a CNN that:

1. Uses multiple kernel sizes ( $3 \times 3$ ,  $5 \times 5$ ,  $7 \times 7$ ) in parallel
2. Applies to malware detection with byte sequences
3. Visualizes which kernel sizes detect which patterns
4. Implements attention mechanism to weight kernel outputs
5. Compares performance with single-kernel CNN

## **Exercise 2: Denoising Autoencoder**

Create a denoising autoencoder that:

1. Adds controlled noise to network traffic data
2. Learns to reconstruct clean data from noisy input
3. Uses the decoder as an anomaly detector
4. Implements different noise types (Gaussian, dropout, adversarial)
5. Evaluates robustness to different attack types

## **Exercise 3: Hierarchical Autoencoder**

243

Design a hierarchical system that:

1. First autoencoder clusters similar traffic patterns
2. Specialized autoencoders for each cluster
3. Routes new data to appropriate specialist
4. Detects both cluster-level and within-cluster anomalies
5. Adapts cluster assignments over time

## **Exercise 4: Real-time Log Monitor**

Implement a system that:

1. Processes streaming log data with sliding windows
2. Uses LSTM autoencoder for sequence modeling
3. Maintains normal behavior profiles per user/system
4. Triggers alerts for sustained anomalies
5. Provides interpretable anomaly explanations

## **Exercise 5: Adversarial Robustness**

Create a framework that:

1. Generates adversarial examples for your autoencoder
  2. Trains ensemble with adversarial augmentation
  3. Tests robustness to evasion attacks
  4. Implements defense mechanisms (input validation, ensemble voting)
  5. Measures model stability under attack
-

This section covered advanced architectures for pattern recognition and anomaly detection:

## Key Concepts Mastered:

| Concept                    | Formula/Definition                              | Application              |
|----------------------------|---|--------------------------|
| <b>Convolution</b>         | $(f*g)[n] = \sum f[m]g[n-m]$                    | Pattern detection        |
| <b>Pooling</b>             | max/mean(region)                                | Dimensionality reduction |
| <b>Embedding</b>           | $E: \mathbb{Z} \rightarrow \mathbb{R}^d$        | Discrete to continuous   |
| <b>Reconstruction Loss</b> | $\ X - \hat{X}\ ^2$                             | Anomaly score            |
| <b>Ensemble Voting</b>     | $\sum \text{predictions} \geq \text{threshold}$ | Robust detection         |

## Architecture Selection Guide:

Image/Binary Data → CNN  
Text/Categorical → Embedding + CNN/RNN  
Anomaly Detection → Autoencoder  
High Stakes → Ensemble  
Real-time → Optimized Single Model

245

## Critical Insights:

1. **CNN Design:** Match receptive field to pattern size
2. **Embedding Size:**  $\propto \log(\text{vocabulary\_size})$
3. **Latent Dimension:** 10-20% of input dimension
4. **Ensemble Size:** 3-5 models optimal
5. **Threshold Setting:** Use validation data, not training

## Memory Tricks:

- **CNN:** "SLIP" - Slide, Local, Invariant, Parameters shared
- **Embeddings:** "TED" - Tokens to Embedding Dimensions
- **Autoencoders:** "ELDER" - Encode, Latent, Decode, Error, Reconstruct

## Next Steps:

In Section 6, we'll explore advanced architectures using the Functional API, learn optimization techniques with genetic algorithms, and master production deployment strategies for real-world security applications.

# Machine Learning Training Documentation

## Section 6: Functional Models and Deployment

### Overview

This final section explores advanced model architectures using TensorFlow's Functional API, automated optimization techniques, and production deployment strategies. We'll build complex models with multiple inputs and outputs, master genetic algorithms for hyperparameter tuning, and learn to deploy secure, scalable ML systems. The section culminates with real-world deployment patterns and monitoring strategies.

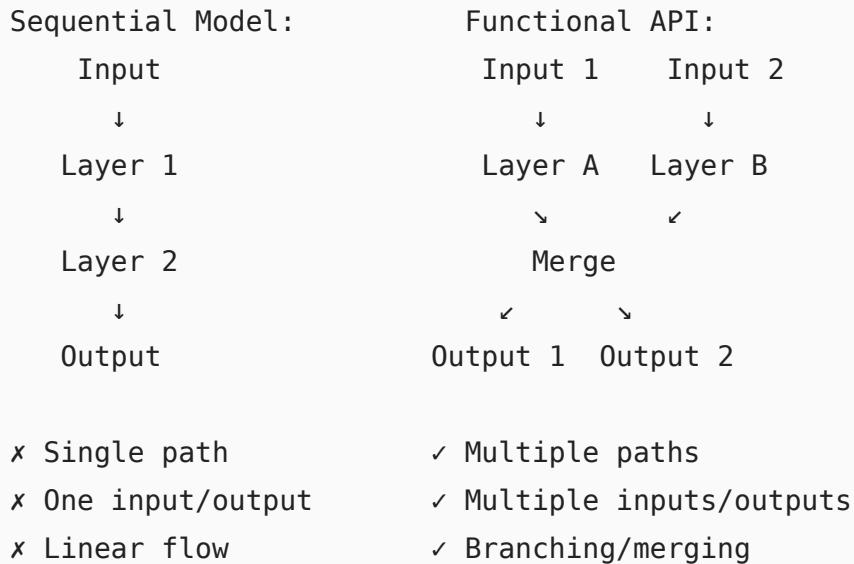
### Learning Objectives

By the end of this section, you will be able to:

- Design complex architectures with the Functional API
- Build models with multiple inputs, outputs, and branches
- Implement data augmentation strategies
- Apply genetic algorithms for hyperparameter optimization
- Deploy models using containers and cloud services
- Monitor and maintain production ML systems

# Why Functional API?

Sequential models have limitations:



247

## Functional API Patterns

| Pattern                     | Use Case                 | Example                     |
|-----------------------------|--------------------------|-----------------------------|
| <b>Multi-Input</b>          | Different data types     | Image + metadata            |
| <b>Multi-Output</b>         | Multiple tasks           | Classification + regression |
| <b>Shared Layers</b>        | Transfer learning        | Feature extractor reuse     |
| <b>Residual Connections</b> | Deep networks            | Skip connections            |
| <b>Attention Mechanisms</b> | Focus on important parts | Transformer models          |

## Mathematical Foundation

Functional Composition:

$$h = f \circ g \circ k$$

$$h(x) = f(g(k(x)))$$

In TensorFlow:

```
output = layer3(layer2(layer1(input)))
```

## Memory Trick: "FORMS"

Flexible Operations, Reusable Modules, Shared layers



# Common Architecture Components

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import numpy as np
import matplotlib.pyplot as plt

# Visualize different architecture patterns
def visualize_architecture_patterns():
    """Show common Functional API patterns"""

    fig, axes = plt.subplots(2, 3, figsize=(15, 10))

    # 1. Multi-Input Architecture
    ax = axes[0, 0]
    ax.set_title('Multi-Input Model', fontsize=14, fontweight='bold')

    # Draw architecture
    # Inputs
    ax.text(0.2, 0.9, 'Image\nInput', ha='center',
bbox=dict(boxstyle="round,pad=0.3", facecolor="lightblue"))
    ax.text(0.2, 0.7, 'Metadata\nInput', ha='center',
bbox=dict(boxstyle="round,pad=0.3", facecolor="lightgreen"))

    # Processing
    ax.text(0.5, 0.9, 'CNN\nBranch', ha='center',
bbox=dict(boxstyle="round,pad=0.3", facecolor="gray"))
    ax.text(0.5, 0.7, 'Dense\nBranch', ha='center',
bbox=dict(boxstyle="round,pad=0.3", facecolor="gray"))

    # Merge
    ax.text(0.7, 0.8, 'Concatenate', ha='center',
bbox=dict(boxstyle="round,pad=0.3", facecolor="yellow"))
    ax.text(0.9, 0.8, 'Output', ha='center',
bbox=dict(boxstyle="round,pad=0.3", facecolor="lightcoral"))

    # Arrows
    ax.arrow(0.25, 0.9, 0.2, 0, head_width=0.02, head_length=0.02,
fc='black')
    ax.arrow(0.25, 0.7, 0.2, 0, head_width=0.02, head_length=0.02,
fc='black')
    ax.arrow(0.55, 0.9, 0.1, -0.08, head_width=0.02, head_length=0.02,
fc='black')
    ax.arrow(0.55, 0.7, 0.1, 0.08, head_width=0.02, head_length=0.02,
fc='black')
    ax.arrow(0.75, 0.8, 0.1, 0, head_width=0.02, head_length=0.02,
fc='black')
```

```

ax.set_xlim(0, 1)
ax.set_ylim(0.6, 1)
ax.axis('off')

# 2. Multi-Output Architecture
ax = axes[0, 1]
ax.set_title('Multi-Output Model', fontsize=14, fontweight='bold')

# Architecture
ax.text(0.1, 0.8, 'Input', ha='center',
bbox=dict(boxstyle="round,pad=0.3", facecolor="lightblue"))
ax.text(0.3, 0.8, 'Shared\nLayers', ha='center',
bbox=dict(boxstyle="round,pad=0.3", facecolor="gray"))
ax.text(0.6, 0.9, 'Class\nHead', ha='center',
bbox=dict(boxstyle="round,pad=0.3", facecolor="yellow"))
ax.text(0.6, 0.7, 'Severity\nHead', ha='center',
bbox=dict(boxstyle="round,pad=0.3", facecolor="yellow"))
ax.text(0.85, 0.9, 'Category', ha='center',
bbox=dict(boxstyle="round,pad=0.3", facecolor="lightcoral"))
ax.text(0.85, 0.7, 'Score', ha='center',
bbox=dict(boxstyle="round,pad=0.3", facecolor="lightgreen"))

# Connections
ax.arrow(0.15, 0.8, 0.1, 0, head_width=0.02, head_length=0.02,
fc='black')
ax.arrow(0.35, 0.8, 0.15, 0.08, head_width=0.02, head_length=0.02,
fc='black')
ax.arrow(0.35, 0.8, 0.15, -0.08, head_width=0.02, head_length=0.02,
fc='black')
ax.arrow(0.65, 0.9, 0.15, 0, head_width=0.02, head_length=0.02,
fc='black')
ax.arrow(0.65, 0.7, 0.15, 0, head_width=0.02, head_length=0.02,
fc='black')

ax.set_xlim(0, 1)
ax.set_ylim(0.6, 1)
ax.axis('off')

# 3. Residual Connection
ax = axes[0, 2]
ax.set_title('Residual Connection', fontsize=14, fontweight='bold')

# Main path
positions = [0.2, 0.4, 0.6, 0.8]
for i, pos in enumerate(positions):
    ax.text(pos, 0.8, f'Layer\n{i}', ha='center',
            bbox=dict(boxstyle="round,pad=0.3", facecolor="gray"))
    if i < len(positions) - 1:
        ax.arrow(pos + 0.05, 0.8, 0.1, 0, head_width=0.02,

```

```

head_length=0.02, fc='black')

# Skip connection
ax.annotate(' ', xy=(0.6, 0.75), xytext=(0.2, 0.75),
            arrowprops=dict(arrowstyle='->', color='red', lw=2))
ax.text(0.4, 0.7, 'Skip Connection', ha='center', color='red')

# Addition
ax.text(0.6, 0.85, '+', fontsize=20, ha='center')

ax.set_xlim(0.1, 0.9)
ax.set_ylim(0.65, 0.95)
ax.axis('off')

# 4. Attention Mechanism
ax = axes[1, 0]
ax.set_title('Attention Mechanism', fontsize=14, fontweight='bold')

# Query, Key, Value
ax.text(0.2, 0.9, 'Query', ha='center',
bbox=dict(boxstyle="round,pad=0.3", facecolor="lightblue"))
ax.text(0.2, 0.7, 'Key', ha='center',
bbox=dict(boxstyle="round,pad=0.3", facecolor="lightgreen"))
ax.text(0.2, 0.5, 'Value', ha='center',
bbox=dict(boxstyle="round,pad=0.3", facecolor="lightyellow"))

# Attention computation
ax.text(0.5, 0.8, 'Attention\nScores', ha='center',
       bbox=dict(boxstyle="round,pad=0.3", facecolor="gray"))
ax.text(0.5, 0.6, 'QxKT/√d', ha='center', fontsize=10)

# Output
ax.text(0.8, 0.7, 'Weighted\nSum', ha='center',
       bbox=dict(boxstyle="round,pad=0.3", facecolor="lightcoral"))

# Arrows
ax.arrow(0.25, 0.9, 0.15, -0.08, head_width=0.02, head_length=0.02,
fc='black')
ax.arrow(0.25, 0.7, 0.15, 0.08, head_width=0.02, head_length=0.02,
fc='black')
ax.arrow(0.55, 0.8, 0.2, -0.08, head_width=0.02, head_length=0.02,
fc='black')
ax.arrow(0.25, 0.5, 0.45, 0.18, head_width=0.02, head_length=0.02,
fc='black')

ax.set_xlim(0.1, 0.9)
ax.set_ylim(0.4, 1)
ax.axis('off')

# 5. Siamese Network

```

```

ax = axes[1, 1]
ax.set_title('Siamese Network', fontsize=14, fontweight='bold')

# Twin inputs
ax.text(0.15, 0.85, 'Input A', ha='center',
bbox=dict(boxstyle="round,pad=0.3", facecolor="lightblue"))
ax.text(0.15, 0.65, 'Input B', ha='center',
bbox=dict(boxstyle="round,pad=0.3", facecolor="lightblue"))

# Shared encoder
ax.text(0.4, 0.75, 'Shared\nEncoder', ha='center',
bbox=dict(boxstyle="round,pad=0.3", facecolor="gray"))
ax.text(0.4, 0.6, '(Same weights)', ha='center', fontsize=8,
style='italic')

# Embeddings
ax.text(0.65, 0.85, 'Embed A', ha='center',
bbox=dict(boxstyle="round,pad=0.3", facecolor="yellow"))
ax.text(0.65, 0.65, 'Embed B', ha='center',
bbox=dict(boxstyle="round,pad=0.3", facecolor="yellow"))

# Distance
ax.text(0.85, 0.75, 'Distance', ha='center',
bbox=dict(boxstyle="round,pad=0.3", facecolor="lightcoral"))

# Connections
ax.arrow(0.2, 0.85, 0.15, -0.08, head_width=0.02, head_length=0.02,
fc='black')
ax.arrow(0.2, 0.65, 0.15, 0.08, head_width=0.02, head_length=0.02,
fc='black')
ax.arrow(0.45, 0.78, 0.15, 0.05, head_width=0.02, head_length=0.02,
fc='black')
ax.arrow(0.45, 0.72, 0.15, -0.05, head_width=0.02, head_length=0.02,
fc='black')
ax.arrow(0.7, 0.85, 0.1, -0.08, head_width=0.02, head_length=0.02,
fc='black')
ax.arrow(0.7, 0.65, 0.1, 0.08, head_width=0.02, head_length=0.02,
fc='black')

ax.set_xlim(0.05, 0.95)
ax.set_ylim(0.55, 0.95)
ax.axis('off')

# 6. Ensemble Architecture
ax = axes[1, 2]
ax.set_title('Ensemble Architecture', fontsize=14, fontweight='bold')

# Input
ax.text(0.1, 0.75, 'Input', ha='center',
bbox=dict(boxstyle="round,pad=0.3", facecolor="lightblue"))

```

```

# Multiple models
models = ['CNN', 'RNN', 'Dense']
colors = ['lightgreen', 'lightyellow', 'lightcoral']
for i, (model, color) in enumerate(zip(models, colors)):
    y_pos = 0.85 - i*0.1
    ax.text(0.4, y_pos, model, ha='center',
            bbox=dict(boxstyle="round, pad=0.3", facecolor=color))
    ax.arrow(0.15, 0.75, 0.2, y_pos-0.75, head_width=0.02,
             head_length=0.02, fc='black')
    ax.arrow(0.45, y_pos, 0.2, 0.75-y_pos, head_width=0.02,
             head_length=0.02, fc='black')

# Aggregation
ax.text(0.7, 0.75, 'Vote/\nAverage', ha='center',
        bbox=dict(boxstyle="round, pad=0.3", facecolor="gray"))
ax.text(0.9, 0.75, 'Output', ha='center',
        bbox=dict(boxstyle="round, pad=0.3", facecolor="lightblue"))
ax.arrow(0.75, 0.75, 0.1, 0, head_width=0.02, head_length=0.02,
         fc='black')

ax.set_xlim(0, 1)
ax.set_ylim(0.5, 0.95)
ax.axis('off')

plt.tight_layout()
plt.show()

visualize_architecture_patterns()

```

```

# Implement various Functional API patterns
class FunctionalModelBuilder:
    """Builder for complex architectures"""

    @staticmethod
    def build_multi_input_model(image_shape, metadata_dim):
        """Model with image and metadata inputs"""

        # Image input branch
        image_input = layers.Input(shape=image_shape, name='image_input')
        x = layers.Conv2D(32, (3, 3), activation='relu')(image_input)
        x = layers.MaxPooling2D((2, 2))(x)
        x = layers.Conv2D(64, (3, 3), activation='relu')(x)
        x = layers.GlobalAveragePooling2D()(x)
        image_features = layers.Dense(128, activation='relu',
                                     name='image_features')(x)

        # Metadata input branch
        metadata_input = layers.Input(shape=(metadata_dim,), name='metadata_input')
        y = layers.Dense(64, activation='relu')(metadata_input)
        y = layers.BatchNormalization()(y)
        metadata_features = layers.Dense(64, activation='relu',
                                         name='metadata_features')(y)

        # Merge branches
        combined = layers.concatenate([image_features, metadata_features], name='combined_features')
        z = layers.Dense(128, activation='relu')(combined)
        z = layers.Dropout(0.5)(z)

        # Output
        output = layers.Dense(1, activation='sigmoid',
                             name='threat_probability')(z)

        model = keras.Model(
            inputs=[image_input, metadata_input],
            outputs=output,
            name='multi_input_threat_detector'
        )

        return model

    @staticmethod
    def build_multi_output_model(input_shape):
        """Model with multiple task outputs"""

        # Shared layers

```

```

        input_layer = layers.Input(shape=input_shape, name='input')
        shared = layers.Dense(256, activation='relu')(input_layer)
        shared = layers.BatchNormalization()(shared)
        shared = layers.Dense(128, activation='relu')(shared)
        shared = layers.Dropout(0.3)(shared)

        # Task-specific heads
        # Classification head
        class_branch = layers.Dense(64, activation='relu',
name='class_branch')(shared)
        class_output = layers.Dense(5, activation='softmax',
name='threat_category')(class_branch)

        # Severity regression head
        severity_branch = layers.Dense(32, activation='relu',
name='severity_branch')(shared)
        severity_output = layers.Dense(1, activation='linear',
name='severity_score')(severity_branch)

        # Confidence head
        confidence_branch = layers.Dense(32, activation='relu',
name='confidence_branch')(shared)
        confidence_output = layers.Dense(1, activation='sigmoid',
name='confidence')(confidence_branch)

    model = keras.Model(
        inputs=input_layer,
        outputs=[class_output, severity_output, confidence_output],
        name='multi_task_analyzer'
    )

    return model

@staticmethod
def build_residual_block(x, filters, kernel_size=3):
    """Residual block with skip connection"""

    # Main path
    y = layers.Conv2D(filters, kernel_size, padding='same')(x)
    y = layers.BatchNormalization()(y)
    y = layers.Activation('relu')(y)
    y = layers.Conv2D(filters, kernel_size, padding='same')(y)
    y = layers.BatchNormalization()(y)

    # Skip connection
    if x.shape[-1] != filters:
        # Adjust dimensions if needed
        x = layers.Conv2D(filters, 1, padding='same')(x)

    # Add skip connection

```

```
        out = layers.Add()([x, y])
        out = layers.Activation('relu')(out)

    return out

# Demonstrate model building
# Multi-input model
multi_input_model = FunctionalModelBuilder.build_multi_input_model(
    image_shape=(64, 64, 3),
    metadata_dim=10
)

# Multi-output model
multi_output_model = FunctionalModelBuilder.build_multi_output_model(
    input_shape=(100, )
)

# Visualize architectures
print("Multi-Input Model:")
keras.utils.plot_model(multi_input_model, show_shapes=True,
show_layer_names=True)

print("\nMulti-Output Model:")
keras.utils.plot_model(multi_output_model, show_shapes=True,
show_layer_names=True)
```

# Why Data Augmentation?

Limited Data → Augmentation → Virtual Samples → Better Generalization  
1000 → × 10 → 10,000 → Robust Model

## Augmentation Techniques

| Type      | Technique   | Parameters          | Use Case              |
|-----------|-------------|---------------------|-----------------------|
| Geometric | Rotation    | ±degrees            | Rotation invariance   |
| Geometric | Translation | ±pixels             | Position invariance   |
| Geometric | Scaling     | factor              | Size invariance       |
| Geometric | Flipping    | horizontal/vertical | Mirror invariance     |
| Pixel     | Brightness  | ±factor             | Lighting conditions   |
| Pixel     | Contrast    | factor              | Visibility variations |
| Pixel     | Noise       | $\sigma$            | Sensor noise          |
| Advanced  | Cutout      | patch size          | Occlusion robustness  |
| Advanced  | Mixup       | $\alpha$            | Smooth predictions    |
| Advanced  | CutMix      | $\lambda$           | Local features        |

257

## Mathematical Formulations

Mixup:

$$\tilde{x} = \lambda x_1 + (1-\lambda)x_2$$

$$\tilde{y} = \lambda y_1 + (1-\lambda)y_2$$

where  $\lambda \sim \text{Beta}(\alpha, \alpha)$

CutMix:

$$\tilde{x} = M \odot x_1 + (1-M) \odot x_2$$

$$\tilde{y} = \lambda y_1 + (1-\lambda)y_2$$

where  $M$  is binary mask,  $\lambda = \text{area}(M)/\text{area}(\text{image})$

## Security-Specific Augmentations

| Domain         | Augmentation  | Purpose               |
|----------------|---------------|-----------------------|
| Malware Images | Byte shifting | Position independence |
| Network Logs   | Time warping  | Temporal variations   |

| Domain       | Augmentation      | Purpose               |
|--------------|-------------------|-----------------------|
| System Calls | Sequence padding  | Length variations     |
| Text Logs    | Token replacement | Vocabulary robustness |

## Memory Trick: "GASP"

Geometric, Appearance, Synthetic, Probabilistic

---

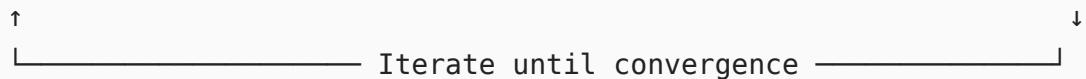
# Why Genetic Algorithms?

Traditional optimization methods have limitations:

| Method                | Pros                          | Cons                   |
|-----------------------|-------------------------------|------------------------|
| Grid Search           | Exhaustive                    | Exponential complexity |
| Random Search         | Simple                        | No learning            |
| Bayesian Optimization | Efficient                     | Complex implementation |
| Genetic Algorithms    | Global search, parallelizable | Slower convergence     |

## Genetic Algorithm Components

Population → Evaluation → Selection → Crossover → Mutation → New Population



## GA Operations

259

| Operation | Purpose            | Example                |
|-----------|--------------------|------------------------|
| Encoding  | Represent solution | [64, 32, 'relu', 0.01] |
| Fitness   | Evaluate quality   | Validation accuracy    |
| Selection | Choose parents     | Tournament, roulette   |
| Crossover | Combine genes      | Single-point, uniform  |
| Mutation  | Add diversity      | Random bit flip        |
| Elitism   | Preserve best      | Top-k carry over       |

## Mathematical Framework

Fitness Function:

$$f(\text{chromosome}) = \text{accuracy} - \lambda \times \text{complexity}$$

Selection Probability (Roulette):

$$P(i) = f(i) / \sum f(j)$$

Crossover (Single-point):

$$\text{child1} = \text{parent1}[:k] + \text{parent2}[k:]$$

```

child2 = parent2[:k] + parent1[k:]

Mutation:
gene_new = gene_old + N(0, σ²) if random() < p_mutate

```

## Implementation

```

import random
from typing import List, Dict, Tuple

class GeneticOptimizer:
    """Genetic algorithm for neural network hyperparameter optimization"""

    def __init__(self, param_ranges: Dict, population_size: int = 50):
        self.param_ranges = param_ranges
        self.population_size = population_size
        self.generation = 0

    def create_individual(self) -> Dict:
        """Create random individual"""
        individual = {}
        for param, (min_val, max_val, param_type) in
        self.param_ranges.items():
            if param_type == 'int':
                individual[param] = random.randint(min_val, max_val)
            elif param_type == 'float':
                individual[param] = random.uniform(min_val, max_val)
            elif param_type == 'choice':
                individual[param] = random.choice(min_val) # min_val is
list of choices
        return individual

    def create_population(self) -> List[Dict]:
        """Initialize population"""
        return [self.create_individual() for _ in
range(self.population_size)]

    def evaluate_fitness(self, individual: Dict, X_train, y_train, X_val,
y_val) -> float:
        """Evaluate individual fitness (model performance)"""
        # Build model with hyperparameters
        model = self.build_model(individual)

        # Train briefly
        history = model.fit(
            X_train, y_train,
            validation_data=(X_val, y_val),
            epochs=10, # Quick evaluation

```

```

        batch_size=32,
        verbose=0
    )

    # Fitness = validation accuracy - complexity penalty
    val_accuracy = max(history.history['val_accuracy'])
    complexity = individual['layers'] *
individual['neurons_per_layer'] / 10000
    fitness = val_accuracy - 0.1 * complexity

    return fitness

def build_model(self, params: Dict) -> keras.Model:
    """Build model from genetic encoding"""
    model = keras.Sequential()

    # Input layer
    model.add(layers.Input(shape=(params['input_dim'],)))

    # Hidden layers
    for i in range(params['layers']):
        model.add(layers.Dense(
            params['neurons_per_layer'],
            activation=params['activation']
        ))
        if params['use_dropout']:
            model.add(layers.Dropout(params['dropout_rate']))
        if params['use_batch_norm']:
            model.add(layers.BatchNormalization())

    # Output layer
    model.add(layers.Dense(params['output_dim'],
activation='softmax'))

    # Compile
    model.compile(
        optimizer=keras.optimizers.Adam(learning_rate=params['learning_rate']),
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy']
    )

    return model

def tournament_selection(self, population: List[Tuple[Dict, float]],
                       tournament_size: int = 3) -> Dict:
    """Select parent via tournament"""
    tournament = random.sample(population, tournament_size)
    winner = max(tournament, key=lambda x: x[1])
    return winner[0]

```

```

    def crossover(self, parent1: Dict, parent2: Dict) -> Tuple[Dict, Dict]:
        """Single-point crossover"""
        child1, child2 = {}, {}
        crossover_point = random.randint(1, len(parent1) - 1)

        params = list(parent1.keys())
        for i, param in enumerate(params):
            if i < crossover_point:
                child1[param] = parent1[param]
                child2[param] = parent2[param]
            else:
                child1[param] = parent2[param]
                child2[param] = parent1[param]

        return child1, child2

    def mutate(self, individual: Dict, mutation_rate: float = 0.1) -> Dict:
        """Mutate individual genes"""
        mutated = individual.copy()

        for param in mutated:
            if random.random() < mutation_rate:
                # Re-randomize this parameter
                min_val, max_val, param_type = self.param_ranges[param]
                if param_type == 'int':
                    mutated[param] = random.randint(min_val, max_val)
                elif param_type == 'float':
                    # Small perturbation
                    delta = random.gauss(0, (max_val - min_val) * 0.1)
                    mutated[param] = max(min_val, min(max_val,
                        mutated[param] + delta))
                elif param_type == 'choice':
                    mutated[param] = random.choice(min_val)

        return mutated

    def evolve(self, X_train, y_train, X_val, y_val, generations: int = 20):
        """Run genetic algorithm"""
        # Initialize population
        population = self.create_population()
        best_individuals = []

        for gen in range(generations):
            print(f"\nGeneration {gen + 1}/{generations}")

            # Evaluate fitness

```

```

        fitness_scores = []
        for individual in population:
            fitness = self.evaluate_fitness(individual, X_train,
y_train, X_val, y_val)
            fitness_scores.append((individual, fitness))

        # Sort by fitness
        fitness_scores.sort(key=lambda x: x[1], reverse=True)
best_individuals.append(fitness_scores[0])

print(f"Best fitness: {fitness_scores[0][1]:.4f}")
print(f"Best params: {fitness_scores[0][0]")

# Create new population
new_population = []

# Elitism: keep top 10%
elite_size = self.population_size // 10
new_population.extend([ind for ind, _ in
fitness_scores[:elite_size]])

# Generate rest through crossover and mutation
while len(new_population) < self.population_size:
    parent1 = self.tournament_selection(fitness_scores)
    parent2 = self.tournament_selection(fitness_scores)

    child1, child2 = self.crossover(parent1, parent2)
    child1 = self.mutate(child1)
    child2 = self.mutate(child2)

    new_population.extend([child1, child2])

population = new_population[:self.population_size]

return best_individuals

# Visualize genetic algorithm process
def visualize_ga_evolution():
    """Visualize GA optimization process"""

    # Simulate GA evolution
generations = 20
population_size = 50

    # Simulate fitness evolution
best_fitness = []
avg_fitness = []
diversity = []

    for gen in range(generations):

```

```

# Simulate improvement
gen_best = 0.6 + 0.3 * (1 - np.exp(-gen/5)) + random.gauss(0,
0.02)
gen_avg = 0.4 + 0.2 * (1 - np.exp(-gen/5)) + random.gauss(0, 0.03)
gen_diversity = 0.8 * np.exp(-gen/10) + 0.2

best_fitness.append(gen_best)
avg_fitness.append(gen_avg)
diversity.append(gen_diversity)

fig, axes = plt.subplots(2, 2, figsize=(12, 10))

# 1. Fitness evolution
ax = axes[0, 0]
ax.plot(best_fitness, 'g-', linewidth=2, label='Best')
ax.plot(avg_fitness, 'b--', linewidth=2, label='Average')
ax.fill_between(range(generations), avg_fitness, best_fitness,
alpha=0.3)
ax.set_xlabel('Generation')
ax.set_ylabel('Fitness')
ax.set_title('Fitness Evolution')
ax.legend()
ax.grid(True, alpha=0.3)

# 2. Diversity
ax = axes[0, 1]
ax.plot(diversity, 'r-', linewidth=2)
ax.fill_between(range(generations), 0, diversity, alpha=0.3,
color='red')
ax.set_xlabel('Generation')
ax.set_ylabel('Population Diversity')
ax.set_title('Genetic Diversity Over Time')
ax.grid(True, alpha=0.3)

# 3. Parameter landscape
ax = axes[1, 0]
# Simulate 2D parameter space
x = np.linspace(0, 10, 100)
y = np.linspace(0, 10, 100)
X, Y = np.meshgrid(x, y)
Z = np.sin(X/2) * np.cos(Y/2) + 0.1 * X # Fitness landscape

contour = ax.contourf(X, Y, Z, levels=20, cmap='viridis')

# Show GA path
ga_path_x = [2 + i*0.3 + random.gauss(0, 0.2) for i in range(20)]
ga_path_y = [2 + i*0.3 + random.gauss(0, 0.2) for i in range(20)]
ax.plot(ga_path_x, ga_path_y, 'r-', linewidth=2, marker='o',
markersize=4)
ax.set_xlabel('Parameter 1')

```

```

ax.set_ylabel('Parameter 2')
ax.set_title('GA Navigation in Parameter Space')
plt.colorbar(contour, ax=ax, label='Fitness')

# 4. Chromosome visualization
ax = axes[1, 1]
ax.set_title('Chromosome Evolution', fontsize=14)

# Show chromosome structure
chromosome_labels = ['Layers', 'Neurons', 'LR', 'Dropout',
'Activation']
n_genes = len(chromosome_labels)

# Show 3 generations
for gen_idx, gen in enumerate([0, 10, 19]):
    y_offset = gen_idx * 0.3

    # Draw chromosome
    for i in range(n_genes):
        # Simulate gene values
        if i < 2: # Integer genes
            value = random.randint(2, 8)
            color = plt.cm.Blues(value / 8)
        elif i == 2: # Float gene
            value = random.uniform(0.001, 0.1)
            color = plt.cm.Greens(value / 0.1)
        elif i == 3: # Float gene
            value = random.uniform(0.1, 0.5)
            color = plt.cm.Oranges(value / 0.5)
        else: # Choice gene
            value = random.choice(['relu', 'tanh', 'elu'])
            color = plt.cm.Purples(0.5)

        rect = plt.Rectangle((i, y_offset), 0.8, 0.2,
                           facecolor=color, edgecolor='black')
        ax.add_patch(rect)

        if gen_idx == 0:
            ax.text(i + 0.4, -0.1, chromosome_labels[i],
                    ha='center', rotation=45, fontsize=8)

    ax.text(-0.5, y_offset + 0.1, f'Gen {gen}', ha='right',
    fontsize=10)

    ax.set_xlim(-1, n_genes)
    ax.set_ylim(-0.2, 1)
    ax.axis('off')

plt.tight_layout()
plt.show()

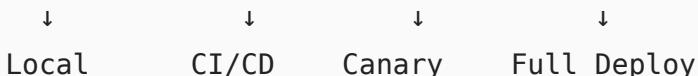
```

```
visualize_ga_evolution()
```

---

# Deployment Pipeline

Development → Testing → Staging → Production



## Deployment Strategies

| Strategy          | Description                        | Risk     | Rollback Speed |
|-------------------|------------------------------------|----------|----------------|
| <b>Blue-Green</b> | Switch entire environment          | Low      | Instant        |
| <b>Canary</b>     | Gradual rollout                    | Very Low | Fast           |
| <b>Rolling</b>    | Update instances sequentially      | Medium   | Moderate       |
| <b>Shadow</b>     | Test with real traffic (no impact) | None     | N/A            |

## Container Architecture

### Application Layer

- └ Model Server (TensorFlow Serving)
- └ API Gateway (FastAPI)
- └ Load Balancer (NGINX)
- └ Monitoring (Prometheus)

267

### Infrastructure Layer

- └ Container Runtime (Docker)
- └ Orchestration (Kubernetes)
- └ Service Mesh (Istio)
- └ Cloud Platform (AWS/GCP/Azure)

## Model Serving Options

| Option           | Pros              | Cons              | Best For      |
|------------------|-------------------|-------------------|---------------|
| <b>REST API</b>  | Simple, universal | Higher latency    | Web apps      |
| <b>gRPC</b>      | Fast, efficient   | Complex setup     | Microservices |
| <b>Batch</b>     | High throughput   | Not real-time     | Analytics     |
| <b>Streaming</b> | Real-time         | Complex           | Live data     |
| <b>Edge</b>      | Low latency       | Limited resources | IoT           |

## **Memory Trick: "SCALE"**

**Serve, Containerize, Automate, Load balance, Evaluate**

---

# Key Metrics to Monitor

| Category    | Metric              | Alert Threshold | Action           |
|-------------|---------------------|-----------------|------------------|
| Performance | Latency             | > 100ms (p95)   | Scale up         |
| Performance | Throughput          | < 1000 req/s    | Add replicas     |
| Accuracy    | Prediction drift    | > 5% drop       | Retrain          |
| Resource    | CPU usage           | > 80%           | Scale out        |
| Resource    | Memory              | > 90%           | Optimize model   |
| Business    | False positive rate | > 10%           | Adjust threshold |

## Model Drift Detection

Types of Drift:

1. Concept Drift:  $P(y|X)$  changes
2. Data Drift:  $P(X)$  changes
3. Label Drift:  $P(y)$  changes

Detection Methods:

- Statistical tests (KS, Chi-square)
- Distribution monitoring
- Performance tracking
- A/B testing

269

## Production Monitoring Implementation

```
import time
import numpy as np
from datetime import datetime
import matplotlib.pyplot as plt
from collections import deque

class ModelMonitor:
    """Production model monitoring system"""

    def __init__(self, window_size=1000):
        self.window_size = window_size
        self.predictions = deque(maxlen=window_size)
        self.latencies = deque(maxlen=window_size)
        self.features = deque(maxlen=window_size)
        self.timestamps = deque(maxlen=window_size)
```

```

# Baseline statistics (from training)
self.baseline_stats = {}

# Alert thresholds
self.thresholds = {
    'latency_p95': 100, # ms
    'accuracy_drop': 0.05,
    'drift_score': 0.1,
    'error_rate': 0.1
}

def set_baseline(self, X_train, y_train):
    """Establish baseline statistics from training data"""
    self.baseline_stats = {
        'feature_means': np.mean(X_train, axis=0),
        'feature_stds': np.std(X_train, axis=0),
        'label_distribution': np.bincount(y_train) / len(y_train)
    }

def log_prediction(self, features, prediction, latency):
    """Log a single prediction"""
    self.predictions.append(prediction)
    self.latencies.append(latency)
    self.features.append(features)
    self.timestamps.append(datetime.now())

def calculate_drift(self):
    """Calculate feature drift using KS statistic"""
    if len(self.features) < 100:
        return 0.0

    recent_features = np.array(list(self.features)[-100:])
    recent_means = np.mean(recent_features, axis=0)
    recent_stds = np.std(recent_features, axis=0)

    # Normalized difference
    mean_drift = np.mean(np.abs(
        (recent_means - self.baseline_stats['feature_means']) /
        (self.baseline_stats['feature_stds'] + 1e-6)
    ))

    return mean_drift

def get_metrics(self):
    """Calculate current metrics"""
    if len(self.predictions) < 10:
        return None

    metrics = {
        'latency_p50': np.percentile(self.latencies, 50),

```

```

        'latency_p95': np.percentile(self.latencies, 95),
        'latency_p99': np.percentile(self.latencies, 99),
        'prediction_rate': len(self.predictions) /
                            (self.timestamps[-1] -
                             self.timestamps[0]).total_seconds(),
        'drift_score': self.calculate_drift(),
        'prediction_distribution': np.bincount(self.predictions) /
len(self.predictions)
    }

    return metrics

def check_alerts(self):
    """Check if any metrics exceed thresholds"""
    metrics = self.get_metrics()
    if not metrics:
        return []

    alerts = []

    if metrics['latency_p95'] > self.thresholds['latency_p95']:
        alerts.append({
            'type': 'LATENCY',
            'severity': 'HIGH',
            'message': f"P95 latency {metrics['latency_p95']:.1f}ms exceeds threshold",
            'value': metrics['latency_p95']
        })

    if metrics['drift_score'] > self.thresholds['drift_score']:
        alerts.append({
            'type': 'DRIFT',
            'severity': 'MEDIUM',
            'message': f"Feature drift detected: {metrics['drift_score']:.3f}",
            'value': metrics['drift_score']
        })

    return alerts

def visualize_monitoring_dashboard(self):
    """Create monitoring dashboard visualization"""
    metrics = self.get_metrics()
    if not metrics:
        print("Not enough data for visualization")
        return

    fig, axes = plt.subplots(2, 3, figsize=(15, 10))

    # 1. Latency distribution

```

```

ax = axes[0, 0]
ax.hist(self.latencies, bins=50, alpha=0.7, color='blue')
ax.axvline(metrics['latency_p95'], color='red', linestyle='--',
            label=f'P95: {metrics["latency_p95"]:.1f}ms')
ax.set_xlabel('Latency (ms)')
ax.set_ylabel('Count')
ax.set_title('Latency Distribution')
ax.legend()

# 2. Prediction rate over time
ax = axes[0, 1]
# Create time buckets
time_buckets = {}
for ts, pred in zip(self.timestamps, self.predictions):
    bucket = ts.replace(second=0, microsecond=0)
    if bucket not in time_buckets:
        time_buckets[bucket] = 0
    time_buckets[bucket] += 1

times = sorted(time_buckets.keys())
rates = [time_buckets[t] for t in times]

ax.plot(times, rates, 'g-', linewidth=2)
ax.set_xlabel('Time')
ax.set_ylabel('Predictions/minute')
ax.set_title('Prediction Rate')
ax.tick_params(axis='x', rotation=45)

# 3. Feature drift
ax = axes[0, 2]
# Simulate drift over time
drift_scores = []
window = 100
for i in range(window, len(self.features), 10):
    recent = np.array(list(self.features)[i-window:i])
    drift = np.mean(np.std(recent, axis=0))
    drift_scores.append(drift)

ax.plot(drift_scores, 'r-', linewidth=2)
ax.axhline(self.thresholds['drift_score'], color='red',
            linestyle='--', label='Alert threshold')
ax.set_xlabel('Time Window')
ax.set_ylabel('Drift Score')
ax.set_title('Feature Drift Monitoring')
ax.legend()

# 4. Prediction distribution
ax = axes[1, 0]
if hasattr(self, 'baseline_stats') and 'label_distribution' in
self.baseline_stats:

```

```

x = np.arange(len(metrics['prediction_distribution']))
width = 0.35

ax.bar(x - width/2, self.baseline_stats['label_distribution'],
       width, label='Training', alpha=0.7)
ax.bar(x + width/2, metrics['prediction_distribution'],
       width, label='Current', alpha=0.7)

ax.set_xlabel('Class')
ax.set_ylabel('Proportion')
ax.set_title('Prediction Distribution Shift')
ax.legend()

# 5. Alert timeline
ax = axes[1, 1]
alerts = self.check_alerts()
alert_types = [a['type'] for a in alerts]
alert_counts = {t: alert_types.count(t) for t in set(alert_types)}

if alert_counts:
    ax.bar(alert_counts.keys(), alert_counts.values(),
color='orange')
    ax.set_xlabel('Alert Type')
    ax.set_ylabel('Count')
    ax.set_title('Active Alerts')
else:
    ax.text(0.5, 0.5, 'No Active Alerts', ha='center',
va='center',
           transform=ax.transAxes, fontsize=20, color='green')
    ax.axis('off')

# 6. System health
ax = axes[1, 2]
health_metrics = {
    'Latency': 1 - min(metrics['latency_p95'] / 200, 1), # Good
if < 200ms
    'Drift': 1 - min(metrics['drift_score'] / 0.2, 1), # Good
if < 0.2
    'Load': min(metrics['prediction_rate'] / 100, 1), # Good
if > 100/s
}

categories = list(health_metrics.keys())
values = list(health_metrics.values())

# Radar chart
angles = np.linspace(0, 2 * np.pi, len(categories),
endpoint=False).tolist()
values += values[:1]
angles += angles[:1]

```

```

        ax.plot(angles, values, 'o-', linewidth=2, color='green')
        ax.fill(angles, values, alpha=0.25, color='green')
        ax.set_ylim(0, 1)
        ax.set_xticks(angles[:-1])
        ax.set_xticklabels(categories)
        ax.set_title('System Health Score')
        ax.grid(True)

    plt.suptitle('Model Monitoring Dashboard', fontsize=16)
    plt.tight_layout()
    plt.show()

# Demonstrate monitoring
monitor = ModelMonitor()

# Simulate production usage
print("Simulating production monitoring...")
for i in range(500):
    # Simulate prediction
    features = np.random.randn(20)
    prediction = np.random.choice([0, 1, 2, 3, 4], p=[0.4, 0.3, 0.2, 0.05,
0.05])
    latency = np.random.gamma(10, 5) # Realistic latency distribution
    monitor.log_prediction(features, prediction, latency)

    # Check for alerts periodically
    if i % 100 == 0 and i > 0:
        alerts = monitor.check_alerts()
        if alerts:
            print(f"\nTimestamp {i}: ALERTS DETECTED:")
            for alert in alerts:
                print(f" - [{alert['severity']}]: {alert['message']}")

# Visualize dashboard
monitor.visualize_monitoring_dashboard()

```

274

## 6.7 Ethics and Bias in ML Systems

### Overview

Machine learning systems can perpetuate and amplify biases present in data, algorithms, and deployment contexts. This section covers critical ethical considerations for security ML systems, focusing on identifying, measuring, and mitigating various forms of bias.

# Types of Bias in Machine Learning

## 1. Algorithmic Bias

Bias introduced by the algorithm itself or its assumptions:

| Source             | Example                                   | Impact                               | Mitigation  |
|--------------------|---|--------------------------------------|---|
| Feature Selection  | Using zip code for fraud detection        | Discriminates against neighborhoods  | Use domain expertise, test for proxy discrimination |
| Model Architecture | Linear models on non-linear relationships | Systematic errors for certain groups | Use flexible models, validate across subgroups      |
| Objective Function | Optimizing only for accuracy              | Ignores fairness constraints         | Multi-objective optimization                        |
| Hyperparameters    | High regularization                       | Underfits minority classes           | Class-specific tuning                               |

## 2. Training Data Bias

```
class TrainingBiasAnalyzer:  
    """Analyze and visualize training data bias"""  
  
    def __init__(self):  
        self.bias_metrics = []  
  
    def analyze_representation_bias(self, X, y, sensitive_attributes):  
        """Check if all groups are adequately represented"""  
        representation_stats = []  
  
        for attr in sensitive_attributes:  
            unique_values, counts = np.unique(X[:, attr],  
return_counts=True)  
            representation_stats[attr] = {  
                'values': unique_values,  
                'counts': counts,  
                'proportions': counts / len(X),  
                'min_representation': np.min(counts) / len(X)  
            }  
  
        # Check class balance within groups  
        for val in unique_values:  
            mask = X[:, attr] == val  
            class_dist = np.bincount(y[mask]) / np.sum(mask)  
            representation_stats[attr][f'class_dist_{val}'] =
```

```

class_dist

        self.bias_metrics['representation'] = representation_stats
    return representation_stats

def analyze_label_bias(self, y, sensitive_groups):
    """Analyze bias in labels across sensitive groups"""
    label_bias_stats = {}

    for group_name, group_mask in sensitive_groups.items():
        positive_rate = np.mean(y[group_mask])
        negative_rate = 1 - positive_rate

        label_bias_stats[group_name] = {
            'positive_rate': positive_rate,
            'negative_rate': negative_rate,
            'sample_size': np.sum(group_mask)
        }

    # Calculate disparate impact
    rates = [stats['positive_rate'] for stats in
label_bias_stats.values()]
    if len(rates) >= 2:
        disparate_impact = min(rates) / max(rates)
        label_bias_stats['disparate_impact'] = disparate_impact

    self.bias_metrics['labels'] = label_bias_stats
    return label_bias_stats

def visualize_bias(self):
    """Visualize various forms of bias"""
    fig, axes = plt.subplots(2, 2, figsize=(12, 10))

    # 1. Representation bias
    ax = axes[0, 0]
    if 'representation' in self.bias_metrics:
        rep_data = self.bias_metrics['representation']
        for attr, stats in rep_data.items():
            if isinstance(stats, dict) and 'proportions' in stats:
                ax.bar(stats['values'], stats['proportions'],
alpha=0.7, label=f'Attribute {attr}')
            ax.set_ylabel('Proportion')
            ax.set_title('Representation Bias')
            ax.legend()

    # 2. Label bias
    ax = axes[0, 1]
    if 'labels' in self.bias_metrics:
        label_data = self.bias_metrics['labels']
        groups = [k for k in label_data.keys() if k !=

```

```

'disparate_impact']
    positive_rates = [label_data[g]['positive_rate'] for g in
groups]

    ax.bar(groups, positive_rates, color=['red', 'blue', 'green'][:len(groups)])
    ax.axhline(y=np.mean(positive_rates), color='black',
linestyle='--', label='Average')
    ax.set_ylabel('Positive Class Rate')
    ax.set_title('Label Bias Across Groups')
    ax.legend()

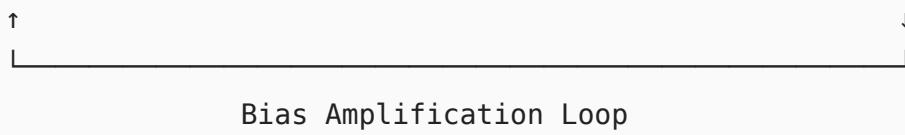
    if 'disparate_impact' in label_data:
        ax.text(0.5, 0.95, f'Disparate Impact:
{label_data["disparate_impact"]:.3f}',
                transform=ax.transAxes, ha='center')

plt.tight_layout()
plt.show()

```

### 3. Feedback Loop Bias

Initial Model → Predictions → Actions → New Data → Updated Model



277

### Feedback Loop Examples in Security

| System            | Feedback Loop                            | Bias Amplification                            | Mitigation                        |
|-------------------|--|---|-----------------------------------|
| Fraud Detection   | Flagged users investigated more          | Increases detection rate for flagged groups   | Random sampling for investigation |
| Threat Scoring    | High-risk areas get more monitoring      | Confirms initial bias through more detections | Normalize by monitoring intensity |
| Access Control    | Denied users can't prove trustworthiness | Perpetuates exclusion                         | Probationary access periods       |
| Anomaly Detection | Rare behaviors flagged as anomalous      | Penalizes minorities                          | Cluster-specific thresholds       |

### Measuring Fairness

```

class FairnessMetrics:
    """Calculate various fairness metrics"""

```

```

    @staticmethod
    def demographic_parity(y_pred, sensitive_attr):
        """
        Demographic parity:  $P(\hat{y}=1|A=0) = P(\hat{y}=1|A=1)$ 
        """
        groups = np.unique(sensitive_attr)
        positive_rates = []

        for group in groups:
            mask = sensitive_attr == group
            rate = np.mean(y_pred[mask])
            positive_rates.append(rate)

        disparity = max(positive_rates) - min(positive_rates)
        return {
            'positive_rates': dict(zip(groups, positive_rates)),
            'max_disparity': disparity,
            'satisfied': disparity < 0.1 # 10% threshold
        }

    @staticmethod
    def equal_opportunity(y_true, y_pred, sensitive_attr):
        """
        Equal opportunity:  $P(\hat{y}=1|y=1,A=0) = P(\hat{y}=1|y=1,A=1)$ 
        True positive rates should be equal
        """
        groups = np.unique(sensitive_attr)
        tpr_values = []

        for group in groups:
            mask = sensitive_attr == group
            group_y_true = y_true[mask]
            group_y_pred = y_pred[mask]

            # True positive rate
            positive_mask = group_y_true == 1
            if np.sum(positive_mask) > 0:
                tpr = np.mean(group_y_pred[positive_mask])
                tpr_values.append(tpr)

        disparity = max(tpr_values) - min(tpr_values) if tpr_values else 0
        return {
            'tpr_values': dict(zip(groups, tpr_values)),
            'max_disparity': disparity,
            'satisfied': disparity < 0.1
        }

    @staticmethod
    def equalized_odds(y_true, y_pred, sensitive_attr):

```

```

"""
Equalized odds: Both TPR and FPR should be equal across groups
"""

equal_opp = FairnessMetrics.equal_opportunity(y_true, y_pred,
sensitive_attr)

# Also check false positive rates
groups = np.unique(sensitive_attr)
fpr_values = []

for group in groups:
    mask = sensitive_attr == group
    group_y_true = y_true[mask]
    group_y_pred = y_pred[mask]

    # False positive rate
    negative_mask = group_y_true == 0
    if np.sum(negative_mask) > 0:
        fpr = np.mean(group_y_pred[negative_mask])
        fpr_values.append(fpr)

fpr_disparity = max(fpr_values) - min(fpr_values) if fpr_values
else 0

return {
    'tpr_disparity': equal_opp['max_disparity'],
    'fpr_disparity': fpr_disparity,
    'satisfied': equal_opp['satisfied'] and fpr_disparity < 0.1
}

```

279

## Bias Mitigation Strategies

### Pre-processing: Fix the Data

```

class BiasedDataMitigation:
    """Pre-processing techniques to reduce bias"""

    @staticmethod
    def resampling(X, y, sensitive_attr, strategy='oversample'):
        """Balance representation across sensitive groups"""
        from imblearn.over_sampling import SMOTE
        from imblearn.under_sampling import RandomUnderSampler

        if strategy == 'oversample':
            # Oversample minority groups
            sampler = SMOTE(random_state=42)
        else:
            # Undersample majority groups

```

```

        sampler = RandomUnderSampler(random_state=42)

        # Create synthetic attribute combining class and sensitive
        attribute
        combined = y * 10 + sensitive_attr
        X_resampled, combined_resampled = sampler.fit_resample(X,
combined)

        # Separate back
        y_resampled = combined_resampled // 10
        sensitive_resampled = combined_resampled % 10

        return X_resampled, y_resampled, sensitive_resampled

    @staticmethod
    def feature_transformation(X, sensitive_indices):
        """Remove correlation with sensitive attributes"""
        # Create a copy
        X_fair = X.copy()

        # For each non-sensitive feature
        for i in range(X.shape[1]):
            if i not in sensitive_indices:
                # Remove correlation with sensitive attributes
                for s_idx in sensitive_indices:
                    # Linear regression to find correlation
                    correlation = np.corrcoef(X[:, i], X[:, s_idx])[0, 1]
                    if abs(correlation) > 0.1: # If correlated
                        # Orthogonalize
                        X_fair[:, i] = X[:, i] - correlation * X[:, s_idx]

        return X_fair

```

280

## In-processing: Fair Learning Algorithms

```

class FairClassifier:
    """Classifier with fairness constraints"""

    def __init__(self, base_model,
fairness_constraint='demographic_parity', lambda_fair=1.0):
        self.base_model = base_model
        self.fairness_constraint = fairness_constraint
        self.lambda_fair = lambda_fair

    def fit(self, X, y, sensitive_attr):
        """Train with fairness constraints"""
        if self.fairness_constraint == 'demographic_parity':
            # Custom loss with fairness penalty

```

```

    def fair_loss(y_true, y_pred):
        # Standard loss
        base_loss = keras.losses.binary_crossentropy(y_true,
y_pred)

        # Fairness penalty
        groups = tf.unique(sensitive_attr)[0]
        positive_rates = []
        for group in groups:
            mask = tf.equal(sensitive_attr, group)
            group_pred = tf.boolean_mask(y_pred, mask)
            positive_rates.append(tf.reduce_mean(group_pred))

        fairness_penalty = tf.math.reduce_variance(positive_rates)

        return base_loss + self.lambda_fair * fairness_penalty

    self.base_model.compile(optimizer='adam', loss=fair_loss)

    return self.base_model.fit(X, y)

```

## Post-processing: Adjust Predictions

```

class FairPostProcessor:
    """Post-process predictions to ensure fairness"""

    def __init__(self, fairness_metric='equal_opportunity'):
        self.fairness_metric = fairness_metric
        self.group_thresholds = {}

    def fit(self, y_true, y_scores, sensitive_attr):
        """Find optimal thresholds per group"""
        from sklearn.metrics import roc_curve

        groups = np.unique(sensitive_attr)

        if self.fairness_metric == 'equal_opportunity':
            # Find thresholds that equalize TPR
            target_tpr = 0.8 # Target true positive rate

            for group in groups:
                mask = sensitive_attr == group
                fpr, tpr, thresholds = roc_curve(y_true[mask],
y_scores[mask])

                # Find threshold closest to target TPR
                idx = np.argmin(np.abs(tpr - target_tpr))
                self.group_thresholds[group] = thresholds[idx]

```

281

```

        return self

    def predict(self, y_scores, sensitive_attr):
        """Apply group-specific thresholds"""
        y_pred = np.zeros_like(y_scores)

        for group, threshold in self.group_thresholds.items():
            mask = sensitive_attr == group
            y_pred[mask] = (y_scores[mask] >= threshold).astype(int)

    return y_pred

```

## Ethical Guidelines for Security ML

### 1. Transparency

- Document data sources and potential biases
- Explain model decisions, especially for high-stakes applications
- Provide audit trails for predictions

### 2. Accountability

- Regular bias audits
- Clear escalation paths for disputed decisions
- Human oversight for critical actions

### 3. Privacy

- Minimize collection of sensitive attributes
- Use differential privacy when possible
- Regular data retention reviews

### 4. Continuous Monitoring

```

class BiasMonitor:
    """Continuous monitoring for bias in production"""

    def __init__(self, window_size=1000):
        self.window_size = window_size
        self.prediction_buffer = []
        self.alerts = []

    def update(self, predictions, sensitive_attrs, timestamps):
        """Update monitoring with new predictions"""
        self.prediction_buffer.extend(zip(predictions,
                                          sensitive_attrs, timestamps))

        # Keep only recent predictions
        if len(self.prediction_buffer) > self.window_size:
            self.prediction_buffer = self.prediction_buffer[-]

```

```

        self.window_size:]

        # Check for bias
        self._check_bias_drift()

    def _check_bias_drift(self):
        """Check if bias metrics are drifting"""
        if len(self.prediction_buffer) < 100:
            return

        predictions = np.array([p[0] for p in self.prediction_buffer])
        sensitive = np.array([p[1] for p in self.prediction_buffer])

        # Calculate current bias
        groups = np.unique(sensitive)
        positive_rates = []
        for group in groups:
            mask = sensitive == group
            rate = np.mean(predictions[mask])
            positive_rates.append(rate)

        max_disparity = max(positive_rates) - min(positive_rates)

        if max_disparity > 0.15: # Alert threshold
            self.alerts.append({
                'timestamp': self.prediction_buffer[-1][2],
                'metric': 'demographic_parity',
                'value': max_disparity,
                'severity': 'high' if max_disparity > 0.25 else
    'medium'
            })

```

283

## Best Practices Summary

| Practice                 | Implementation                    | Benefit                |
|--------------------------|-----------------------------------|------------------------|
| <b>Diverse Teams</b>     | Include ethicists, domain experts | Catch blind spots      |
| <b>Bias Testing</b>      | Automated fairness tests in CI/CD | Early detection        |
| <b>Documentation</b>     | Bias impact assessments           | Accountability         |
| <b>Regular Audits</b>    | Quarterly fairness reviews        | Continuous improvement |
| <b>Feedback Channels</b> | User reporting mechanisms         | Real-world validation  |

## Memory Trick: "FATE"

Fairness metrics, Audit regularly, Transparency always, Ethical guidelines



## **Exercise 1: Multi-Modal Security Model**

Build a model that:

1. Combines network traffic, system logs, and user behavior
2. Uses different architectures for each input type
3. Implements attention mechanisms between modalities
4. Predicts both threat level and attack type
5. Provides interpretable feature importance

## **Exercise 2: Advanced Data Augmentation**

Create an augmentation system that:

1. Learns which augmentations improve model performance
2. Applies adversarial perturbations for robustness
3. Generates synthetic training data
4. Balances augmentation strength with label preservation
5. Adapts augmentation strategy during training

## **Exercise 3: AutoML for Security**

285

Implement an AutoML system that:

1. Searches architecture space (layers, units, activations)
2. Optimizes preprocessing pipelines
3. Selects best loss functions and metrics
4. Implements neural architecture search (NAS)
5. Provides Pareto frontier of accuracy vs efficiency

## **Exercise 4: Production-Ready API**

Build a complete API that:

1. Handles multiple model versions
2. Implements A/B testing for models
3. Provides real-time monitoring dashboard
4. Auto-scales based on load
5. Includes comprehensive logging and alerting

## **Exercise 5: Federated Learning System**

Design a system that:

1. Trains models across distributed data sources
  2. Preserves privacy of local data
  3. Aggregates model updates securely
  4. Handles non-IID data distributions
  5. Detects and mitigates poisoning attacks
-

This final section covered advanced topics in model development and deployment:

## Key Concepts Mastered:

| Concept                   | Key Points                          | Application             |
|---------------------------|-------------------------------------|-------------------------|
| <b>Functional API</b>     | Multi-input/output, shared layers   | Complex architectures   |
| <b>Data Augmentation</b>  | Geometric, pixel, advanced          | Improved generalization |
| <b>Genetic Algorithms</b> | Global optimization, parallelizable | Hyperparameter tuning   |
| <b>Deployment</b>         | Containers, orchestration, serving  | Production systems      |
| <b>Monitoring</b>         | Metrics, drift, alerts              | Model maintenance       |

## Architecture Decision Framework:

### 1. Problem Complexity?

Simple → Sequential API

Complex → Functional API

### 2. Data Availability?

Limited → Heavy augmentation

Abundant → Light augmentation

287

### 3. Optimization Budget?

Low → Random/Grid search

High → Genetic algorithms

### 4. Deployment Scale?

Small → REST API

Large → Kubernetes + gRPC

### 5. Monitoring Needs?

Basic → Logs + metrics

Advanced → Full observability

## Production Checklist:

- Model:** Versioned, documented, tested
- API:** Rate limited, authenticated, validated
- Container:** Optimized size, security scanned
- Deployment:** Blue-green or canary strategy
- Monitoring:** Metrics, logs, alerts configured

- Documentation:** API docs, runbooks, architecture
- Security:** Input validation, model robustness
- Compliance:** Data privacy, audit trails

## Memory Tricks:

- **Functional API:** "FORMS" - Flexible Operations, Reusable Modules, Shared
- **Augmentation:** "GASP" - Geometric, Appearance, Synthetic, Probabilistic
- **GA:** "FSCME" - Fitness, Selection, Crossover, Mutation, Elitism
- **Deployment:** "SCALE" - Serve, Containerize, Automate, Load balance, Evaluate

## Final Thoughts:

You've now mastered the complete machine learning pipeline from data acquisition to production deployment. The key to success in real-world applications is:

1. **Start Simple:** Baseline models first
2. **Iterate Quickly:** Fast feedback loops
3. **Monitor Everything:** You can't improve what you don't measure
4. **Plan for Failure:** Systems will fail, be ready
5. **Keep Learning:** The field evolves rapidly

Remember: The best model is not the most complex one, but the one that reliably solves your problem in production!

# Exercise Solutions

## Section 1: Data Acquisition Exercise Solutions

### Exercise 1: Security Log Parser

**Task:** Build a robust security log parser that handles multiple formats.

```
import re
import pandas as pd
from datetime import datetime
import json

class SecurityLogParser:
    """
        Comprehensive security log parser for multiple formats
    """

    def __init__(self):
        # Define regex patterns for different log formats
        self.patterns = {
            'apache': re.compile(
                r'(?P<ip>\d+\.\d+\.\d+\.\d+) - - \[(?P<timestamp>[\^\]]+)\] '
                "(?P<method>\w+) (?P<path>[^ ]+) [^"]+" "(?P<status>\d+) (?P<size>\d+|-)' ),
            'nginx': re.compile(
                r'(?P<ip>\d+\.\d+\.\d+\.\d+) - - \[(?P<timestamp>[\^\]]+)\] '
                "(?P<method>\w+) (?P<path>[^ ]+) [^"]+" "(?P<status>\d+) (?P<size>\d+) "(?P<referrer>[^"]*)" "(?P<user_agent>[^"]*)"'),
            'syslog': re.compile(
                r'(?P<timestamp>\w+ \d+ \d+:\d+:\d+) (?P<hostname>\S+) (?P<service>\S+?) (?:(\[(?P<pid>\d+)\])?: (?P<message>.*))'),
            'windows_security': re.compile(
                r'(?P<timestamp>\d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2}) (?P<event_id>\d+) (?P<level>\w+) (?P<source>\S+) (?P<message>.*)'),
            'firewall': re.compile(
                r'(?P<timestamp>\d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2}) (?P<action>\w+) (?P<protocol>\w+) (?P<src_ip>\d+\.\d+\.\d+\.\d+):(?P<src_port>\d+) -> (?P<dst_ip>\d+\.\d+\.\d+\.\d+):(?P<dst_port>\d+)')
        }

        # Date format mappings
        self.date_formats = {
```

```

'apache': '%d/%b/%Y:%H:%M:%S %z',
'nginx': '%d/%b/%Y:%H:%M:%S %z',
'syslog': '%b %d %H:%M:%S',
'windows_security': '%Y-%m-%d %H:%M:%S',
'firewall': '%Y-%m-%d %H:%M:%S'
}

self.parsed_logs = []

def detect_format(self, log_line):
    """Automatically detect log format"""
    for format_name, pattern in self.patterns.items():
        if pattern.match(log_line):
            return format_name
    return None

def parse_line(self, log_line, format_type=None):
    """Parse a single log line"""
    # Auto-detect format if not specified
    if not format_type:
        format_type = self.detect_format(log_line)

    if not format_type:
        return None

    pattern = self.patterns.get(format_type)
    match = pattern.match(log_line)

    if match:
        parsed = match.groupdict()

        # Parse timestamp
        if 'timestamp' in parsed:
            try:
                # Handle syslog special case (no year)
                if format_type == 'syslog':
                    current_year = datetime.now().year
                    timestamp_str = f'{current_year}{parsed["timestamp"]}'
                    parsed['timestamp'] = datetime.strptime(
                        timestamp_str, f"%Y{self.date_formats[format_type]}")
                else:
                    parsed['timestamp'] = datetime.strptime(
                        parsed['timestamp'],
                        self.date_formats[format_type])
            except:
                # Keep original if parsing fails

```

```
    pass

    # Convert numeric fields
    for field in ['status', 'size', 'pid', 'event_id', 'src_port',
'dst_port']:
        if field in parsed and parsed[field] and parsed[field] !=
'-':
            try:
                parsed[field] = int(parsed[field])
            except:
                pass

    parsed['log_format'] = format_type
    return parsed

return None

def parse_file(self, file_path, format_type=None):
    """Parse entire log file"""
    parsed_logs = []
    errors = []

    try:
        with open(file_path, 'r', encoding='utf-8', errors='ignore')
as f:
            for line_num, line in enumerate(f, 1):
                line = line.strip()
                if not line:
                    continue

                try:
                    parsed = self.parse_line(line, format_type)
                    if parsed:
                        parsed['line_number'] = line_num
                        parsed['source_file'] = file_path
                        parsed_logs.append(parsed)
                    else:
                        errors.append({
                            'line_number': line_num,
                            'content': line,
                            'error': 'Failed to parse'
                        })
                except Exception as e:
                    errors.append({
                        'line_number': line_num,
                        'content': line,
                        'error': str(e)
                    })

    except Exception as e:
```

```
        print(f"Error reading file: {e}")
        return None, None

    self.parsed_logs.extend(parsed_logs)

    # Convert to DataFrame
    df = pd.DataFrame(parsed_logs)

    return df, errors

def extract_security_events(self, df=None):
    """Extract security-relevant events"""
    if df is None:
        df = pd.DataFrame(self.parsed_logs)

    security_events = []

    # Check for failed authentications (HTTP 401, 403)
    if 'status' in df.columns:
        failed_auth = df[df['status'].isin([401, 403])]
        for _, row in failed_auth.iterrows():
            security_events.append({
                'event_type': 'failed_authentication',
                'timestamp': row.get('timestamp'),
                'source_ip': row.get('ip') or row.get('src_ip'),
                'details': row.to_dict()
            })

    # Check for suspicious user agents
    if 'user_agent' in df.columns:
        suspicious_agents = ['scanner', 'bot', 'crawler', 'nikto',
        'sqlmap']
        mask =
        df['user_agent'].str.lower().str.contains('|'.join(suspicious_agents),
        na=False)
        suspicious = df[mask]
        for _, row in suspicious.iterrows():
            security_events.append({
                'event_type': 'suspicious_user_agent',
                'timestamp': row.get('timestamp'),
                'source_ip': row.get('ip'),
                'user_agent': row.get('user_agent'),
                'details': row.to_dict()
            })

    # Check for port scans (multiple ports from same IP)
    if 'src_ip' in df.columns and 'dst_port' in df.columns:
        port_scan_threshold = 10
        ip_port_counts = df.groupby('src_ip')['dst_port'].nunique()
        suspicious_ips = ip_port_counts[ip_port_counts >
        292]
```

```

port_scan_threshold].index

        for ip in suspicious_ips:
            ip_data = df[df['src_ip'] == ip]
            security_events.append({
                'event_type': 'potential_port_scan',
                'timestamp': ip_data['timestamp'].min(),
                'source_ip': ip,
                'unique_ports': int(ip_port_counts[ip]),
                'details': {
                    'targeted_ports':
ip_data['dst_port'].unique().tolist()
                }
            })

# Check for high-frequency requests (potential DDoS)
if 'ip' in df.columns and 'timestamp' in df.columns:
    df_time = df.copy()
    df_time['minute'] =
pd.to_datetime(df_time['timestamp']).dt.floor('T')
    req_per_minute = df_time.groupby(['ip', 'minute']).size()
    high_freq = req_per_minute[req_per_minute > 100]

    for (ip, minute), count in high_freq.items():
        security_events.append({
            'event_type': 'high_frequency_requests',
            'timestamp': minute,
            'source_ip': ip,
            'request_count': int(count),
            'details': {'requests_per_minute': int(count)}
        })

```

293

```

return pd.DataFrame(security_events)

def generate_report(self, output_file='security_report.json'):
    """Generate comprehensive security report"""
    df = pd.DataFrame(self.parsed_logs)
    security_events = self.extract_security_events(df)

    report = {
        'summary': {
            'total_logs_parsed': len(self.parsed_logs),
            'time_range': {
                'start': str(df['timestamp'].min()) if 'timestamp' in
df.columns else None,
                'end': str(df['timestamp'].max()) if 'timestamp' in
df.columns else None
            },
            'log_formats_detected':
df['log_format'].value_counts().to_dict() if 'log_format' in df.columns
        }
    }

```

```

        else {},
            'security_events_found': len(security_events)
        },
        'security_events': security_events.to_dict('records') if not
security_events.empty else [],
        'statistics': {}
    }

    # Add format-specific statistics
    if 'status' in df.columns:
        report['statistics']['http_status_codes'] =
df['status'].value_counts().to_dict()

    if 'ip' in df.columns:
        report['statistics']['top_ips'] =
df['ip'].value_counts().head(10).to_dict()

    if 'path' in df.columns:
        report['statistics']['top_paths'] =
df['path'].value_counts().head(10).to_dict()

    # Save report
    with open(output_file, 'w') as f:
        json.dump(report, f, indent=2, default=str)

return report

```

294

```

# Example usage
parser = SecurityLogParser()

# Test with sample logs
sample_logs = [
    '192.168.1.100 - - [01/Jan/2024:12:00:00 +0000] "GET /admin/login.php
HTTP/1.1" 401 5320',
    '10.0.0.50 - - [01/Jan/2024:12:01:00 +0000] "POST /api/user HTTP/1.1"
403 1234 "-" "Mozilla/5.0 (compatible; Nikto/2.1.5)"',
    '2024-01-01 12:05:00 DENY TCP 192.168.1.50:54321 -> 10.0.0.100:22',
    'Jan 01 12:10:15 webserver sshd[1234]: Failed password for root from
192.168.1.200',
]

# Parse individual lines
for log in sample_logs:
    parsed = parser.parse_line(log)
    if parsed:
        print(f"Format: {parsed['log_format']}, Parsed: {parsed}")

# Generate report
report = parser.generate_report()
print("\nSecurity Report Generated!")

```



```
import requests
from bs4 import BeautifulSoup
import asyncio
import aiohttp
from datetime import datetime
import hashlib
import json
import sqlite3
from urllib.parse import urljoin, urlparse
import re

class ThreatIntelligenceScraper:
    """
        Advanced threat intelligence feed scraper with rate limiting and
        caching
    """

    def __init__(self, db_path='threat_intel.db'):
        self.session = None
        self.db_path = db_path
        self.init_database()

        # Define threat intelligence sources
        self.sources = {
            'abuse_ch_urlhaus': {
                'url': 'https://urlhaus.abuse.ch/downloads/text/',
                'type': 'url_list',
                'parser': self.parse_urlhaus
            },
            'abuse_ch_sslbl': {
                'url':
'https://sslbl.abuse.ch/blacklist/sslipblacklist.txt',
                'type': 'ip_list',
                'parser': self.parse_ip_list
            },
            'emerging_threats': {
                'url':
'https://rules.emergingthreats.net/blockrules/compromised-ips.txt',
                'type': 'ip_list',
                'parser': self.parse_ip_list
            },
            'phishtank': {
                'url': 'http://data.phishtank.com/data/online-valid.json',
                'type': 'json',
                'parser': self.parse_phishtank
            }
        }

        # Rate limiting
```

```
    self.rate_limit = asyncio.Semaphore(5) # Max 5 concurrent
requests
    self.request_delay = 1 # Seconds between requests to same domain
    self.last_request_time = {}

def init_database(self):
    """Initialize SQLite database for caching"""
    conn = sqlite3.connect(self.db_path)
    cursor = conn.cursor()

    # Create tables
    cursor.execute('''
        CREATE TABLE IF NOT EXISTS threat_indicators (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            indicator_type TEXT,
            indicator_value TEXT UNIQUE,
            source TEXT,
            confidence REAL,
            first_seen TIMESTAMP,
            last_seen TIMESTAMP,
            metadata TEXT
        )
    ''')
    cursor.execute('''
        CREATE TABLE IF NOT EXISTS scraping_history (
            source TEXT PRIMARY KEY,
            last_scraped TIMESTAMP,
            items_found INTEGER,
            hash TEXT
        )
    ''')

    conn.commit()
    conn.close()

async def rate_limited_request(self, url):
    """Make rate-limited HTTP request"""
    domain = urlparse(url).netloc

    async with self.rate_limit:
        # Check rate limit for domain
        if domain in self.last_request_time:
            elapsed = datetime.now() - self.last_request_time[domain]
            if elapsed.total_seconds() < self.request_delay:
                await asyncio.sleep(self.request_delay -
elapsed.total_seconds())

    try:
        async with self.session.get(url, timeout=30) as response:
```

```

        self.last_request_time[domain] = datetime.now()
        return await response.text()
    except Exception as e:
        print(f"Error fetching {url}: {e}")
        return None

def parse_urlhaus(self, content):
    """Parse URLhaus format"""
    indicators = []
    lines = content.strip().split('\n')

    for line in lines:
        if line.startswith('#') or not line.strip():
            continue

        # URLhaus format: URL
        url = line.strip()
        if url:
            indicators.append({
                'type': 'url',
                'value': url,
                'confidence': 0.8
            })

    return indicators
298

def parse_ip_list(self, content):
    """Parse simple IP list format"""
    indicators = []
    ip_pattern = re.compile(r'^\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}$')

    for line in content.strip().split('\n'):
        if line.startswith('#') or not line.strip():
            continue

        ip = line.split()[0] # Some lists have additional info
        if ip_pattern.match(ip):
            indicators.append({
                'type': 'ip',
                'value': ip,
                'confidence': 0.7
            })

    return indicators
800

def parse_phishtank(self, content):
    """Parse PhishTank JSON format"""
    indicators = []

    try:

```

```

        data = json.loads(content)
        for entry in data:
            if entry.get('verified') == 'yes':
                indicators.append({
                    'type': 'url',
                    'value': entry.get('url'),
                    'confidence': 0.9,
                    'metadata': {
                        'phish_id': entry.get('phish_id'),
                        'target': entry.get('target')
                    }
                })
        except json.JSONDecodeError:
            print("Failed to parse PhishTank JSON")

    return indicators

def parse_html_ioc_page(self, url, content):
    """Parse HTML page for IoCs"""
    indicators = []
    soup = BeautifulSoup(content, 'html.parser')

    # Extract IPs
    ip_pattern = re.compile(r'\b(?:[0-9]{1,3}\.){3}[0-9]{1,3}\b')
    ips = ip_pattern.findall(str(soup))
    for ip in set(ips):
        # Validate IP
        parts = ip.split('.')
        if all(0 <= int(part) <= 255 for part in parts):
            indicators.append({
                'type': 'ip',
                'value': ip,
                'confidence': 0.5
            })

    # Extract URLs
    url_pattern = re.compile(r'https?://[^<>]+')
    urls = url_pattern.findall(str(soup))
    for url in set(urls):
        indicators.append({
            'type': 'url',
            'value': url,
            'confidence': 0.5
        })

    # Extract hashes (MD5, SHA1, SHA256)
    hash_patterns = {
        'md5': re.compile(r'\b[a-fA-F0-9]{32}\b'),
        'sha1': re.compile(r'\b[a-fA-F0-9]{40}\b'),
        'sha256': re.compile(r'\b[a-fA-F0-9]{64}\b')
    }

```



```
# Store indicators
new_count = 0
for indicator in indicators:
    try:
        cursor.execute('''
            INSERT OR REPLACE INTO threat_indicators
                (indicator_type, indicator_value, source, confidence,
first_seen, last_seen, metadata)
            VALUES (?, ?, ?, ?, ?, ?, ?, ?)
        ''', (
            indicator['type'],
            indicator['value'],
            source_name,
            indicator.get('confidence', 0.5),
            datetime.now(),
            datetime.now(),
            json.dumps(indicator.get('metadata', {})))
    ))
    new_count += 1
except sqlite3.IntegrityError:
    # Update last_seen for existing indicator
    cursor.execute('''
        UPDATE threat_indicators
        SET last_seen = ?, confidence = MAX(confidence, ?)
        WHERE indicator_value = ?
    ''', (datetime.now(), indicator.get('confidence', 0.5),
indicator['value']))

# Update scraping history
cursor.execute('''
    INSERT OR REPLACE INTO scraping_history (source, last_scraped,
items_found, hash)
    VALUES (?, ?, ?, ?)
''', (source_name, datetime.now(), new_count, content_hash))

conn.commit()
conn.close()

print(f"{source_name}: Added/updated {new_count} indicators")
return new_count

async def scrape_all(self):
    """Scrape all configured sources"""
    async with aiohttp.ClientSession() as session:
        self.session = session

    tasks = [
        self.scrape_source(name, config)
        for name, config in self.sources.items()
    ]
```

```
        results = await asyncio.gather(*tasks)

        total_indicators = sum(results)
        print(f"\nTotal indicators collected: {total_indicators}")

    def search_indicators(self, query, indicator_type=None):
        """Search for indicators in database"""
        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()

        if indicator_type:
            cursor.execute('''
                SELECT * FROM threat_indicators
                WHERE indicator_value LIKE ? AND indicator_type = ?
                ORDER BY confidence DESC
            ''', (f'%{query}%', indicator_type))
        else:
            cursor.execute('''
                SELECT * FROM threat_indicators
                WHERE indicator_value LIKE ?
                ORDER BY confidence DESC
            ''', (f'%{query}%',))

        results = cursor.fetchall()
        conn.close()

        return results

    def export_indicators(self, output_file='threat_indicators.json',
                           min_confidence=0.5, indicator_type=None):
        """Export indicators to file"""
        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()

        query = "SELECT * FROM threat_indicators WHERE confidence >= ?"
        params = [min_confidence]

        if indicator_type:
            query += " AND indicator_type = ?"
            params.append(indicator_type)

        cursor.execute(query, params)

        indicators = []
        for row in cursor.fetchall():
            indicators.append({
                'type': row[1],
                'value': row[2],
                'source': row[3],
```

```
        'confidence': row[4],
        'first_seen': row[5],
        'last_seen': row[6],
        'metadata': json.loads(row[7]) if row[7] else {}
    })

conn.close()

with open(output_file, 'w') as f:
    json.dump(indicators, f, indent=2, default=str)

print(f"Exported {len(indicators)} indicators to {output_file}")
return indicators

def get_statistics(self):
    """Get statistics about collected indicators"""
    conn = sqlite3.connect(self.db_path)
    cursor = conn.cursor()

    # Overall statistics
    cursor.execute("SELECT COUNT(*) FROM threat_indicators")
    total_indicators = cursor.fetchone()[0]

    # By type
    cursor.execute('''
        SELECT indicator_type, COUNT(*)
        FROM threat_indicators
        GROUP BY indicator_type
    ''')
    by_type = dict(cursor.fetchall())

    # By source
    cursor.execute('''
        SELECT source, COUNT(*)
        FROM threat_indicators
        GROUP BY source
    ''')
    by_source = dict(cursor.fetchall())

    # Recent activity
    cursor.execute('''
        SELECT COUNT(*)
        FROM threat_indicators
        WHERE last_seen > datetime('now', '-1 day')
    ''')
    recent_24h = cursor.fetchone()[0]

    conn.close()

    return {
```

```
'total_indicators': total_indicators,
'by_type': by_type,
'by_source': by_source,
'recent_24h': recent_24h
}

# Example usage
async def main():
    scraper = ThreatIntelligenceScraper()

    # Scrape all sources
    await scraper.scrape_all()

    # Get statistics
    stats = scraper.get_statistics()
    print("\nStatistics:")
    print(json.dumps(stats, indent=2))

    # Search for specific indicators
    results = scraper.search_indicators('malware', indicator_type='url')
    print(f"\nFound {len(results)} URL indicators containing 'malware'")

    # Export high-confidence indicators
    scraper.export_indicators(min_confidence=0.7)

# Run the scraper
# asyncio.run(main())
```

```
import pandas as pd
import numpy as np
from datetime import datetime, timedelta
import json
import sqlite3
from kafka import KafkaProducer, KafkaConsumer
import redis
import threading
import queue
import time

class MultiSourceDataPipeline:
    """
    Real-time data pipeline that combines multiple security data sources
    """

    def __init__(self):
        # Data sources configuration
        self.sources = {
            'network_logs': {
                'type': 'streaming',
                'format': 'json',
                'fields': ['timestamp', 'src_ip', 'dst_ip', 'protocol',
                           'bytes']
            },
            'endpoint_logs': {
                'type': 'batch',
                'format': 'csv',
                'fields': ['timestamp', 'hostname', 'process', 'action',
                           'user']
            },
            'threat_intel': {
                'type': 'api',
                'format': 'json',
                'fields': ['indicator', 'type', 'confidence', 'source']
            }
        }

        # Initialize components
        self.redis_client = redis.Redis(host='localhost', port=6379,
                                         decode_responses=True)
        self.data_queue = queue.Queue(maxsize=10000)
        self.enrichment_cache = {}

        # Processing statistics
        self.stats = {
            'events_processed': 0,
            'events_enriched': 0,
            'alerts_generated': 0,
        }
```

```

        'processing_errors': 0
    }

def create_unified_schema(self):
    """Define unified schema for all events"""
    return {
        'event_id': str,
        'timestamp': datetime,
        'source_type': str,
        'severity': str,
        'entity_type': str, # ip, hostname, user, etc.
        'entity_value': str,
        'action': str,
        'details': dict,
        'enrichments': dict,
        'risk_score': float
    }

def normalize_network_log(self, log_entry):
    """Normalize network log to unified schema"""
    try:
        return {
            'event_id':
f"net_{log_entry.get('timestamp')}_{log_entry.get('src_ip')}",
            'timestamp':
datetime.fromisoformat(log_entry.get('timestamp')),
            'source_type': 'network',
            'severity': self.calculate_network_severity(log_entry),
            'entity_type': 'ip',
            'entity_value': log_entry.get('src_ip'),
            'action': f"{log_entry.get('protocol')} connection",
            'details': {
                'src_ip': log_entry.get('src_ip'),
                'dst_ip': log_entry.get('dst_ip'),
                'protocol': log_entry.get('protocol'),
                'bytes': log_entry.get('bytes', 0)
            },
            'enrichments': {},
            'risk_score': 0.0
        }
    except Exception as e:
        self.stats['processing_errors'] += 1
        return None

def normalize_endpoint_log(self, log_entry):
    """Normalize endpoint log to unified schema"""
    try:
        return {
            'event_id':
f"ep_{log_entry.get('timestamp')}_{log_entry.get('hostname')}"
        }
    except Exception as e:
        self.stats['processing_errors'] += 1
        return None

```

```
'timestamp': pd.to_datetime(log_entry.get('timestamp')),
'source_type': 'endpoint',
'severity': self.calculate_endpoint_severity(log_entry),
'entity_type': 'hostname',
'entity_value': log_entry.get('hostname'),
'action': log_entry.get('action'),
'details': {
    'hostname': log_entry.get('hostname'),
    'process': log_entry.get('process'),
    'user': log_entry.get('user'),
    'action': log_entry.get('action')
},
'enrichments': {},
'risk_score': 0.0
}
except Exception as e:
    self.stats['processing_errors'] += 1
    return None

def calculate_network_severity(self, log_entry):
    """Calculate severity based on network indicators"""
    severity = 'low'

    # Check for suspicious ports
    suspicious_ports = [22, 23, 445, 3389, 1433, 3306]
    if any(port in str(log_entry) for port in suspicious_ports):
        severity = 'medium'

    # Check for large data transfers
    if log_entry.get('bytes', 0) > 10000000: # 1MB
        severity = 'high' if severity == 'medium' else 'medium'

    return severity

def calculate_endpoint_severity(self, log_entry):
    """Calculate severity based on endpoint indicators"""
    severity = 'low'

    # Suspicious processes
    suspicious_processes = ['powershell', 'cmd', 'wmic', 'mshta',
'rundll32']
        if any(proc in log_entry.get('process', '').lower() for proc in
suspicious_processes):
            severity = 'medium'

    # Suspicious actions
    suspicious_actions = ['create', 'modify', 'delete', 'execute']
        if any(action in log_entry.get('action', '').lower() for action in
suspicious_actions):
            if severity == 'medium':
```

```
        severity = 'high'
    else:
        severity = 'medium'

    return severity

def enrich_event(self, event):
    """Enrich event with additional context"""
    entity_value = event['entity_value']
    entity_type = event['entity_type']

    # Check cache first
    cache_key = f"{entity_type}:{entity_value}"
    if cache_key in self.enrichment_cache:
        cached_data = self.enrichment_cache[cache_key]
        if (datetime.now() - cached_data['timestamp']).seconds < 3600:
            # 1 hour cache
            event['enrichments'] = cached_data['data']
            self.stats['events_enriched'] += 1
            return event

    enrichments = {}

    # GeoIP enrichment for IPs
    if entity_type == 'ip':
        enrichments['geo'] = self.get_geoip_info(entity_value)
        enrichments['reputation'] =
self.check_ip_reputation(entity_value)
        enrichments['threat_intel'] =
self.check_threat_intel(entity_value, 'ip')

    # Active Directory enrichment for hostnames
    elif entity_type == 'hostname':
        enrichments['owner'] = self.get_asset_owner(entity_value)
        enrichments['criticality'] =
self.get_asset_criticality(entity_value)
        enrichments['last_patch'] =
self.get_last_patch_date(entity_value)

    # User enrichment
    elif entity_type == 'user':
        enrichments['department'] =
self.get_user_department(entity_value)
        enrichments['risk_profile'] =
self.get_user_risk_profile(entity_value)

    event['enrichments'] = enrichments

    # Cache the enrichment
    self.enrichment_cache[cache_key] = {
```

```

        'timestamp': datetime.now(),
        'data': enrichments
    }

    self.stats['events_enriched'] += 1
    return event

def get_geoip_info(self, ip):
    """Mock GeoIP lookup"""
    # In production, use MaxMind or similar
    return {
        'country': 'US',
        'city': 'New York',
        'latitude': 40.7128,
        'longitude': -74.0060
    }

def check_ip_reputation(self, ip):
    """Check IP reputation"""
    # Check against known bad IPs
    bad_ips = ['192.168.1.100', '10.0.0.50'] # Example

    if ip in bad_ips:
        return {'score': 0.9, 'category': 'malicious'}

    # Check Redis for cached reputation
    reputation = self.redis_client.get(f'reputation:{ip}')
    if reputation:
        return json.loads(reputation)

    # Default reputation
    return {'score': 0.1, 'category': 'unknown'}

def check_threat_intel(self, indicator, indicator_type):
    """Check against threat intelligence"""
    # Query threat intel database
    threats = []

    # Mock threat intel check
    if indicator == '192.168.1.100':
        threats.append({
            'source': 'abuse.ch',
            'confidence': 0.8,
            'threat_type': 'botnet_cc'
        })

    return threats

def get_asset_owner(self, hostname):
    """Get asset owner from CMDB"""

```

```
# Mock CMDB lookup
owners = {
    'web-server-01': 'web-team@company.com',
    'db-server-01': 'database-team@company.com'
}
return owners.get(hostname, 'unknown')

def get_asset_criticality(self, hostname):
    """Get asset criticality rating"""
    # Mock criticality lookup
    if 'prod' in hostname.lower():
        return 'high'
    elif 'dev' in hostname.lower():
        return 'low'
    return 'medium'

def get_last_patch_date(self, hostname):
    """Get last patch date for host"""
    # Mock patch management system
    return (datetime.now() - timedelta(days=np.random.randint(1,
90))).isoformat()

def get_user_department(self, username):
    """Get user department from AD/LDAP"""
    departments = {
        'admin': 'IT',
        'jdoe': 'Finance',
        'asmith': 'HR'
    }
    return departments.get(username, 'unknown')

def get_user_risk_profile(self, username):
    """Calculate user risk profile"""
    # Mock risk calculation
    if username == 'admin':
        return {'level': 'high', 'reason': 'privileged_account'}
    return {'level': 'low', 'reason': 'standard_user'}

def calculate_risk_score(self, event):
    """Calculate overall risk score for event"""
    risk_score = 0.0

    # Base score from severity
    severity_scores = {'low': 0.2, 'medium': 0.5, 'high': 0.8}
    risk_score = severity_scores.get(event['severity'], 0.2)

    # Adjust based on enrichments
    enrichments = event.get('enrichments', {})

    # Reputation score
```

```

        if 'reputation' in enrichments:
            risk_score = max(risk_score,
enrichments['reputation'].get('score', 0))

        # Threat intel matches
        if 'threat_intel' in enrichments and enrichments['threat_intel']:
            risk_score = max(risk_score, 0.8)

        # Asset criticality
        if enrichments.get('criticality') == 'high':
            risk_score *= 1.5

        # User risk
        if enrichments.get('risk_profile', {}).get('level') == 'high':
            risk_score *= 1.3

        # Normalize to 0-1 range
        event['risk_score'] = min(risk_score, 1.0)
        return event

    def correlate_events(self, time_window=300):
        """Correlate events within time window"""
        # Group events by entity and time window
        recent_events = []
        correlations = []

        # Get recent events from queue (this is simplified)
        # In production, use a proper event store

        # Example correlation rules

        # Rule 1: Multiple failed logins followed by success
        failed_logins = [e for e in recent_events
                         if e['action'] == 'login_failed']
        success_logins = [e for e in recent_events
                           if e['action'] == 'login_success']

        for success in success_logins:
            user = success['details'].get('user')
            failed_attempts = [f for f in failed_logins
                               if f['details'].get('user') == user
                               and f['timestamp'] < success['timestamp']]

            if len(failed_attempts) >= 3:
                correlations.append({
                    'rule': 'brute_force_success',
                    'severity': 'high',
                    'events': failed_attempts + [success],
                    'description': f"Successful login after
{len(failed_attempts)} failures"
                })

```

```

        })

# Rule 2: Port scan followed by connection
port_scans = [e for e in recent_events
              if 'port_scan' in e.get('action', '')]
connections = [e for e in recent_events
               if e['source_type'] == 'network']

for scan in port_scans:
    src_ip = scan['entity_value']
    subsequent_connections = [c for c in connections
                               if c['details'].get('src_ip') ==
src_ip
                               and c['timestamp'] >
scan['timestamp']]

    if subsequent_connections:
        correlations.append({
            'rule': 'scan_and_exploit',
            'severity': 'high',
            'events': [scan] + subsequent_connections,
            'description': f"Connection after port scan from
{src_ip}"})
}

return correlations

def generate_alert(self, event_or_correlation):
    """Generate alert for high-risk events or correlations"""
    alert = {
        'alert_id': f"alert_{datetime.now().timestamp()}",
        'timestamp': datetime.now(),
        'severity': event_or_correlation.get('severity', 'medium'),
        'title': '',
        'description': '',
        'recommended_actions': [],
        'evidence': []
    }

    if 'rule' in event_or_correlation: # Correlation
        alert['title'] = f"Correlated Activity:
{event_or_correlation['rule']}"
        alert['description'] = event_or_correlation['description']
        alert['evidence'] = event_or_correlation['events']
    else: # Single event
        alert['title'] = f"High Risk Event:
{event_or_correlation['action']}"
        alert['description'] = f"Risk Score:
{event_or_correlation['risk_score']:.2f}"
        alert['evidence'] = [event_or_correlation]
    }

```

```

# Add recommended actions
if alert['severity'] == 'high':
    alert['recommended_actions'].extend([
        'Investigate immediately',
        'Isolate affected systems',
        'Collect forensic evidence'
    ])
elif alert['severity'] == 'medium':
    alert['recommended_actions'].extend([
        'Review logs for additional activity',
        'Monitor for escalation'
    ])

self.stats['alerts_generated'] += 1
return alert

def process_event_stream(self):
    """Main event processing loop"""
    while True:
        try:
            # Get event from queue
            raw_event = self.data_queue.get(timeout=1)

            # Normalize based on source type
            if raw_event['source'] == 'network':
                event = self.normalize_network_log(raw_event['data'])
            elif raw_event['source'] == 'endpoint':
                event = self.normalize_endpoint_log(raw_event['data'])
            else:
                continue

            if not event:
                continue

            # Enrich event
            event = self.enrich_event(event)

            # Calculate risk score
            event = self.calculate_risk_score(event)

            # Store processed event
            self.store_event(event)

            # Check if alert needed
            if event['risk_score'] > 0.7:
                alert = self.generate_alert(event)
                self.send_alert(alert)

            self.stats['events_processed'] += 1

```

```

        # Periodic correlation check
        if self.stats['events_processed'] % 100 == 0:
            correlations = self.correlate_events()
            for correlation in correlations:
                alert = self.generate_alert(correlation)
                self.send_alert(alert)

    except queue.Empty:
        continue
    except Exception as e:
        print(f"Processing error: {e}")
        self.stats['processing_errors'] += 1

def store_event(self, event):
    """Store processed event"""
    # Store in Redis for real-time access
    event_key = f"event:{event['event_id']}"
    self.redis_client.setex(
        event_key,
        3600, # 1 hour TTL
        json.dumps(event, default=str)
    )

    # Also store in time-series format
    timestamp_key = f"events:{event['timestamp'].strftime('%Y%m%d%H')}"
    self.redis_client.zadd(
        timestamp_key,
        {event['event_id']: event['timestamp'].timestamp()}
    )

def send_alert(self, alert):
    """Send alert to notification systems"""
    # Store alert
    alert_key = f"alert:{alert['alert_id']}"
    self.redis_client.set(alert_key, json.dumps(alert, default=str))

    # In production, integrate with:
    # - Email
    # - Slack/Teams
    # - PagerDuty
    # - SIEM

    print(f"ALERT: {alert['title']} - Severity: {alert['severity']}")

def get_statistics(self):
    """Get pipeline statistics"""
    return {
        'events_processed': self.stats['events_processed'],

```

```

        'events_enriched': self.stats['events_enriched'],
        'alerts_generated': self.stats['alerts_generated'],
        'processing_errors': self.stats['processing_errors'],
        'events_per_second': self.calculate_eps(),
        'queue_size': self.data_queue.qsize(),
        'cache_size': len(self.enrichment_cache)
    }

    def calculate_eps(self):
        """Calculate events per second"""
        # This is simplified - in production, track over time windows
        return 0 # Placeholder

# Example usage
pipeline = MultiSourceDataPipeline()

# Simulate some events
sample_events = [
{
    'source': 'network',
    'data': {
        'timestamp': datetime.now().isoformat(),
        'src_ip': '192.168.1.100',
        'dst_ip': '10.0.0.50',
        'protocol': 'TCP',
        'bytes': 1500000
    }
},
{
    'source': 'endpoint',
    'data': {
        'timestamp': datetime.now().isoformat(),
        'hostname': 'workstation-01',
        'process': 'powershell.exe',
        'action': 'execute',
        'user': 'jdoe'
    }
}
]

# Add events to queue
for event in sample_events:
    pipeline.data_queue.put(event)

# Process events (in production, this would run in a separate thread)
# threading.Thread(target=pipeline.process_event_stream,
# daemon=True).start()

# Get statistics

```

```
print("Pipeline Statistics:")
print(json.dumps(pipeline.get_statistics(), indent=2))
```

---

# Exercise 1: Statistical Attack Pattern Analysis

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from scipy import stats
from datetime import datetime, timedelta
import warnings
warnings.filterwarnings('ignore')

class AttackPatternAnalyzer:
    """
    Comprehensive statistical analysis of attack patterns
    """

    def __init__(self):
        self.attack_data = None
        self.statistical_models = {}
        self.anomaly_thresholds = {}

    def generate_attack_data(self, days=30):
        """Generate synthetic attack data for analysis"""
        np.random.seed(42)

        # Time range
        start_date = datetime.now() - timedelta(days=days)
        timestamps = pd.date_range(start=start_date, end=datetime.now(),
                                   freq='H')

        # Generate different attack types with patterns
        n_hours = len(timestamps)

        # Base rates for different attacks
        attack_types = {
            'brute_force': {
                'base_rate': 10,
                'pattern': 'periodic',
                'peak_hours': [2, 3, 4, 14, 15, 16] # Night and afternoon
            },
            'port_scan': {
                'base_rate': 5,
                'pattern': 'random',
                'burst_probability': 0.1
            },
            'sql_injection': {
                'base_rate': 3,
                'pattern': 'business_hours',
                'peak_hours': list(range(9, 18))
            }
        }

        # Create attack data frame
        attack_data = pd.DataFrame({
            'timestamp': timestamps,
            'attack_type': np.random.choice(list(attack_types.keys()),
                                            size=n_hours),
            'rate': np.random.poisson(attack_types[attack_type]['base_rate'],
                                      size=n_hours),
            'pattern': np.random.choice(['periodic', 'random'],
                                       size=n_hours),
            'peak_hours': np.random.choice(attack_types['brute_force']['peak_hours'],
                                           size=n_hours),
            'burst_probability': np.random.uniform(0, 1, size=n_hours)
        })

        return attack_data
```

```
        },
        'ddos': {
            'base_rate': 2,
            'pattern': 'burst',
            'burst_duration': 4
        },
        'malware': {
            'base_rate': 4,
            'pattern': 'increasing',
            'growth_rate': 0.001
        }
    }
}

data = []

for i, timestamp in enumerate(timestamps):
    hour = timestamp.hour
    day_of_week = timestamp.dayofweek

    for attack_type, config in attack_types.items():
        # Calculate attack count based on pattern
        base_rate = config['base_rate']

        if config['pattern'] == 'periodic':
            if hour in config['peak_hours']:
                count = np.random.poisson(base_rate * 3)
            else:
                count = np.random.poisson(base_rate * 0.5)

        elif config['pattern'] == 'business_hours':
            if hour in config['peak_hours'] and day_of_week < 5:
                count = np.random.poisson(base_rate * 2)
            else:
                count = np.random.poisson(base_rate * 0.3)

        elif config['pattern'] == 'burst':
            if np.random.random() < 0.05: # 5% chance of burst
                count = np.random.poisson(base_rate * 20)
            else:
                count = np.random.poisson(base_rate)

        elif config['pattern'] == 'increasing':
            growth_factor = 1 + (config['growth_rate'] * i)
            count = np.random.poisson(base_rate * growth_factor)

        else: # random
            count = np.random.poisson(base_rate)

    # Add some anomalies
    if np.random.random() < 0.01: # 1% chance of anomaly
```

```

        count *= np.random.randint(5, 20)

        data.append({
            'timestamp': timestamp,
            'attack_type': attack_type,
            'count': count,
            'hour': hour,
            'day_of_week': day_of_week,
            'is_weekend': day_of_week >= 5,
            'source_ip_count': np.random.poisson(count * 0.7) if
count > 0 else 0,
            'target_port_count': np.random.poisson(count * 0.3) if
count > 0 else 0,
            'avg_severity': np.random.uniform(1, 10) if count > 0
else 0
        })
    }

    self.attack_data = pd.DataFrame(data)
    return self.attack_data

def temporal_analysis(self):
    """Analyze temporal patterns in attacks"""
    fig, axes = plt.subplots(2, 2, figsize=(15, 10))

    # Hourly distribution
    ax = axes[0, 0]
    hourly_stats = self.attack_data.groupby(['hour', 'attack_type'])[
['count']].mean().unstack()
    hourly_stats.plot(kind='bar', ax=ax, stacked=True)
    ax.set_title('Average Attacks by Hour of Day')
    ax.set_xlabel('Hour')
    ax.set_ylabel('Average Attack Count')
    ax.legend(bbox_to_anchor=(1.05, 1), loc='upper left')

    # Day of week distribution
    ax = axes[0, 1]
    daily_stats = self.attack_data.groupby(['day_of_week',
'attack_type'])['count'].mean().unstack()
    daily_stats.plot(kind='bar', ax=ax)
    ax.set_title('Average Attacks by Day of Week')
    ax.set_xlabel('Day (0=Monday)')
    ax.set_ylabel('Average Attack Count')
    ax.set_xticklabels(['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat',
'Sun'])

    # Time series decomposition for one attack type
    ax = axes[1, 0]
    # Select most common attack type
    main_attack = self.attack_data.groupby('attack_type')[
['count']].sum().idxmax()

```

```

        ts_data = self.attack_data[self.attack_data['attack_type'] == main_attack].set_index('timestamp')['count']

        # Simple moving averages
        ts_data.plot(ax=ax, alpha=0.5, label='Raw')
        ts_data.rolling(window=24).mean().plot(ax=ax, label='24h MA',
        linewidth=2)
        ts_data.rolling(window=168).mean().plot(ax=ax, label='7d MA',
        linewidth=2)
        ax.set_title(f'Time Series: {main_attack}')
        ax.set_xlabel('Time')
        ax.set_ylabel('Attack Count')
        ax.legend()

        # Autocorrelation
        ax = axes[1, 1]
        from statsmodels.graphics.tsaplots import plot_acf
        plot_acf(ts_data.dropna(), lags=48, ax=ax)
        ax.set_title('Autocorrelation Function')

        plt.tight_layout()
        plt.show()

        # Statistical tests for patterns
        print("Statistical Pattern Tests:")
        print("-" * 50)

        # Test for seasonality (daily pattern)
        daily_pattern = self.attack_data.groupby(['attack_type', 'hour'])[
        'count'].mean()
        for attack_type in self.attack_data['attack_type'].unique():
            attack_hourly = daily_pattern[attack_type]
            # Augmented Dickey-Fuller test
            from statsmodels.tsa.stattools import adfuller
            adf_result = adfuller(attack_hourly)
            print(f"\n{attack_type}:")
            print(f"  ADF Statistic: {adf_result[0]:.4f}")
            print(f"  p-value: {adf_result[1]:.4f}")
            print(f"  Stationary: {'Yes' if adf_result[1] < 0.05 else
            'No'}")

    def distribution_analysis(self):
        """Analyze attack count distributions"""
        fig, axes = plt.subplots(2, 3, figsize=(18, 10))
        axes = axes.flatten()

        attack_types = self.attack_data['attack_type'].unique()

        for i, attack_type in enumerate(attack_types):
            ax = axes[i]

```

```

        data = self.attack_data[self.attack_data['attack_type'] ==
attack_type]['count']

        # Histogram with fitted distributions
        n, bins, patches = ax.hist(data, bins=30, density=True,
alpha=0.7, edgecolor='black')

        # Fit different distributions
        distributions = {
            'poisson': stats.poisson,
            'negative_binomial': stats.nbinom,
            'normal': stats.norm
        }

        best_fit = None
        best_ks_stat = float('inf')

        for dist_name, dist in distributions.items():
            try:
                if dist_name == 'poisson':
                    param = data.mean()
                    fitted_dist = dist(param)
                elif dist_name == 'negative_binomial':
                    # Method of moments for negative binomial
                    mean = data.mean()
                    var = data.var()
                    if var > mean:
                        p = mean / var
                        n = mean * p / (1 - p)
                        fitted_dist = dist(n, p)
                    else:
                        continue
                else: # normal
                    param = dist.fit(data)
                    fitted_dist = dist(*param)

                # KS test
                ks_stat, ks_pval = stats.kstest(data, fitted_dist.cdf)

                if ks_stat < best_ks_stat:
                    best_ks_stat = ks_stat
                    best_fit = (dist_name, fitted_dist, ks_pval)

            except:
                continue

        # Plot best fit
        if best_fit:
            x = np.linspace(0, data.max(), 100)
            ax.plot(x, best_fit[1].pdf(x), 'r-', linewidth=2,

```

```

label=f'{best_fit[0]} (p={best_fit[2]:.3f})')

ax.set_title(f'{attack_type} Distribution')
ax.set_xlabel('Attack Count')
ax.set_ylabel('Density')
ax.legend()

# Store distribution parameters
self.statistical_models[attack_type] = best_fit

# Hide unused subplots
for i in range(len(attack_types), len(axes)):
    axes[i].set_visible(False)

plt.tight_layout()
plt.show()

# Print distribution summary
print("\nDistribution Analysis Summary:")
print("-" * 60)
print(f"{'Attack Type':<20} {'Best Fit':<15} {'KS p-value':<10}")
{'Parameters'})
print("-" * 60)

for attack_type, model_info in self.statistical_models.items():
    if model_info:
        dist_name, fitted_dist, p_val = model_info
        print(f"{attack_type:<20} {dist_name:<15} {p_val:<10.4f}",
end=" ")
    if dist_name == 'poisson':
        print(f"\u03bb={fitted_dist.args[0]:.2f}")
    elif dist_name == 'normal':
        print(f"\u03bc={fitted_dist.mean():.2f}, \u03c3=",
{fitted_dist.std():.2f})
    else:
        print(f"{fitted_dist.args}")

def correlation_analysis(self):
    """Analyze correlations between attack types and features"""
    # Pivot data for correlation analysis
    pivot_data = self.attack_data.pivot_table(
        index='timestamp',
        columns='attack_type',
        values='count',
        fill_value=0
    )

    # Calculate correlation matrix
    corr_matrix = pivot_data.corr()

```

```
# Visualize correlations
fig, axes = plt.subplots(1, 2, figsize=(15, 6))

# Heatmap
ax = axes[0]
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', center=0,
            square=True, ax=ax, fmt='.2f')
ax.set_title('Attack Type Correlations')

# Network graph of correlations
ax = axes[1]
# Only show significant correlations
threshold = 0.3
significant_corr = corr_matrix.where(
    (corr_matrix > threshold) | (corr_matrix < -threshold)
)

# Create network visualization
import networkx as nx
G = nx.Graph()

for i in range(len(corr_matrix)):
    for j in range(i+1, len(corr_matrix)):
        if not np.isnan(significant_corr.iloc[i, j]):
            G.add_edge(
                corr_matrix.index[i],
                corr_matrix.columns[j],
                weight=abs(significant_corr.iloc[i, j])
            )

if G.edges():
    pos = nx.spring_layout(G)
    nx.draw_networkx_nodes(G, pos, ax=ax, node_size=3000,
                           node_color='lightblue',
                           edgecolors='black')
    nx.draw_networkx_labels(G, pos, ax=ax, font_size=10)

    # Draw edges with width based on correlation strength
    for edge in G.edges(data=True):
        nx.draw_networkx_edges(G, pos, [(edge[0], edge[1])],
                               ax=ax,
                               width=edge[2]['weight']*5,
                               alpha=0.6)

    ax.set_title('Significant Attack Correlations')
    ax.axis('off')

plt.tight_layout()
plt.show()
```

```

# Time-lagged correlations
print("\nTime-Lagged Correlations:")
print("-" * 50)

for lag in [1, 6, 12, 24]: # 1h, 6h, 12h, 24h lags
    lagged_corr = pivot_data.apply(lambda x:
x.corr(pivot_data.shift(lag)))

# Find strongest lagged correlations
strong_lags = []
for col in lagged_corr.columns:
    for idx in lagged_corr.index:
        if col != idx:
            corr_val = lagged_corr.loc[idx, col]
            if abs(corr_val) > 0.3:
                strong_lags.append((idx, col, corr_val))

if strong_lags:
    print(f"\nLag {lag} hours:")
    for source, target, corr in sorted(strong_lags,
                                         key=lambda x: abs(x[2]),
                                         reverse=True)[:5]:
        print(f"  {source} → {target}: {corr:.3f}")

```

324

```

def anomaly_detection(self):
    """Detect anomalies using multiple statistical methods"""
    fig, axes = plt.subplots(2, 2, figsize=(15, 10))

    # Method 1: Z-score
    ax = axes[0, 0]
    for attack_type in self.attack_data['attack_type'].unique():
        data = self.attack_data[self.attack_data['attack_type'] ==
attack_type]
        z_scores = np.abs(stats.zscore(data['count']))
        anomalies = data[z_scores > 3]

        ax.scatter(data['timestamp'], data['count'], alpha=0.5, s=10)
        ax.scatter(anomalies['timestamp'], anomalies['count'],
                   color='red', s=50, marker='x', label=f'{attack_type}
anomalies')

    ax.set_title('Z-Score Anomaly Detection')
    ax.set_xlabel('Time')
    ax.set_ylabel('Attack Count')
    ax.legend()

    # Method 2: IQR
    ax = axes[0, 1]
    for attack_type in self.attack_data['attack_type'].unique():

```

```

        data = self.attack_data[self.attack_data['attack_type'] == attack_type]
        Q1 = data['count'].quantile(0.25)
        Q3 = data['count'].quantile(0.75)
        IQR = Q3 - Q1
        lower_bound = Q1 - 1.5 * IQR
        upper_bound = Q3 + 1.5 * IQR

        anomalies = data[(data['count'] < lower_bound) | (data['count'] > upper_bound)]

        ax.scatter(data['timestamp'], data['count'], alpha=0.5, s=10)
        ax.scatter(anomalies['timestamp'], anomalies['count'],
                   color='orange', s=50, marker='s')

        ax.set_title('IQR Anomaly Detection')
        ax.set_xlabel('Time')
        ax.set_ylabel('Attack Count')

# Method 3: Isolation Forest
ax = axes[1, 0]
from sklearn.ensemble import IsolationForest

# Prepare features
features = self.attack_data[['count', 'hour', 'day_of_week',
                             'source_ip_count',
                             'target_port_count']]

# Fit Isolation Forest
iso_forest = IsolationForest(contamination=0.05, random_state=42)
anomaly_labels = iso_forest.fit_predict(features)

anomalies = self.attack_data[anomaly_labels == -1]
normal = self.attack_data[anomaly_labels == 1]

ax.scatter(normal['timestamp'], normal['count'], alpha=0.5, s=10,
           label='Normal')
ax.scatter(anomalies['timestamp'], anomalies['count'],
           color='purple', s=50, marker='D', label='Anomaly')

ax.set_title('Isolation Forest Anomaly Detection')
ax.set_xlabel('Time')
ax.set_ylabel('Attack Count')
ax.legend()

# Method 4: EWMA Control Charts
ax = axes[1, 1]
main_attack = self.attack_data.groupby('attack_type')[['count']].sum().idxmax()
data = self.attack_data[self.attack_data['attack_type'] ==

```

```

main_attack].copy()

        # Calculate EWMA
        alpha = 0.2
        data['ewma'] = data['count'].ewm(alpha=alpha).mean()
        data['ewma_std'] = data['count'].ewm(alpha=alpha).std()

        # Control limits
        data['ucl'] = data['ewma'] + 3 * data['ewma_std']
        data['lcl'] = data['ewma'] - 3 * data['ewma_std']

        ax.plot(data['timestamp'], data['count'], 'b-', alpha=0.5,
label='Actual')
        ax.plot(data['timestamp'], data['ewma'], 'g-', linewidth=2,
label='EWMA')
        ax.fill_between(data['timestamp'], data['lcl'], data['ucl'],
alpha=0.2, color='red', label='Control Limits')

        # Mark out-of-control points
        ooc = data[(data['count'] > data['ucl']) | (data['count'] <
data['lcl'])]
        ax.scatter(ooc['timestamp'], ooc['count'], color='red', s=50,
marker='o', label='Out of Control')

        ax.set_title(f'EWMA Control Chart: {main_attack}')
        ax.set_xlabel('Time')
        ax.set_ylabel('Attack Count')
        ax.legend()

plt.tight_layout()
plt.show()

# Summary of anomalies
print("\nAnomaly Detection Summary:")
print("-" * 50)
total_points = len(self.attack_data)

methods = {
    'Z-Score':
len(self.attack_data[np.abs(stats.zscore(self.attack_data['count'])) >
3]),
    'IQR': sum([(data['count'] < data['count'].quantile(0.25) -
1.5 * (data['count'].quantile(0.75) - data['count'].quantile(0.25))) |
(data['count'] > data['count'].quantile(0.75) + 1.5 *
(data['count'].quantile(0.75) - data['count'].quantile(0.25)))]
for _, data in
self.attack_data.groupby('attack_type')),
    'Isolation Forest': len(anomalies),
    'EWMA': len(ooc)
}

```

```
for method, count in methods.items():
    percentage = (count / total_points) * 100
    print(f'{method:<20}: {count:>5} anomalies
({percentage:>5.2f}%)')

def predictive_analysis(self):
    """Build predictive models for attack forecasting"""
    from sklearn.ensemble import RandomForestRegressor
    from sklearn.metrics import mean_squared_error,
mean_absolute_error

        # Prepare time series features
        main_attack = self.attack_data.groupby('attack_type')
['count'].sum().idxmax()
        ts_data = self.attack_data[self.attack_data['attack_type'] ==
main_attack].copy()
        ts_data = ts_data.sort_values('timestamp')

        # Create features
        ts_data['hour_sin'] = np.sin(2 * np.pi * ts_data['hour'] / 24)
        ts_data['hour_cos'] = np.cos(2 * np.pi * ts_data['hour'] / 24)
        ts_data['dow_sin'] = np.sin(2 * np.pi * ts_data['day_of_week'] /
7)
        ts_data['dow_cos'] = np.cos(2 * np.pi * ts_data['day_of_week'] /
7)

        # Lag features
        for lag in [1, 6, 12, 24]:
            ts_data[f'lag_{lag}'] = ts_data['count'].shift(lag)

        # Rolling statistics
        for window in [6, 12, 24]:
            ts_data[f'rolling_mean_{window}'] =
ts_data['count'].rolling(window).mean()
            ts_data[f'rolling_std_{window}'] =
ts_data['count'].rolling(window).std()

        # Drop NaN values
        ts_data = ts_data.dropna()

        # Prepare features and target
        feature_cols = [col for col in ts_data.columns
                        if col not in ['timestamp', 'attack_type',
'count']]
        X = ts_data[feature_cols]
        y = ts_data['count']

        # Train-test split (time-based)
        split_point = int(len(ts_data) * 0.8)
```

```

X_train, X_test = X[:split_point], X[split_point:]
y_train, y_test = y[:split_point], y[split_point:]

# Train model
rf_model = RandomForestRegressor(n_estimators=100,
random_state=42)
rf_model.fit(X_train, y_train)

# Predictions
y_pred = rf_model.predict(X_test)

# Evaluate
mse = mean_squared_error(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)

# Visualize predictions
fig, axes = plt.subplots(2, 1, figsize=(15, 10))

# Actual vs Predicted
ax = axes[0]
test_timestamps = ts_data.iloc[split_point:]['timestamp']
ax.plot(test_timestamps, y_test.values, 'b-', label='Actual',
alpha=0.7)
ax.plot(test_timestamps, y_pred, 'r-', label='Predicted',
alpha=0.7)
ax.fill_between(test_timestamps, y_pred - mae, y_pred + mae,
alpha=0.2, color='red', label='MAE Band')
ax.set_title(f'Attack Forecast: {main_attack}')
ax.set_xlabel('Time')
ax.set_ylabel('Attack Count')
ax.legend()

# Feature importance
ax = axes[1]
feature_importance = pd.DataFrame({
    'feature': feature_cols,
    'importance': rf_model.feature_importances_
}).sort_values('importance', ascending=False).head(10)

ax.barh(feature_importance['feature'],
feature_importance['importance'])
ax.set_xlabel('Importance')
ax.set_title('Top 10 Feature Importances')
ax.invert_yaxis()

plt.tight_layout()
plt.show()

print("\nPredictive Model Performance:")
print(f"MSE: {mse:.2f}")

```

```
        print(f"MAE: {mae:.2f}")
        print(f"RMSE: {np.sqrt(mse):.2f}")

    def generate_report(self):
        """Generate comprehensive statistical report"""
        report = {
            'summary_statistics': {},
            'temporal_patterns': {},
            'correlations': {},
            'anomalies': {},
            'predictions': {}
        }

        # Summary statistics
        for attack_type in self.attack_data['attack_type'].unique():
            data = self.attack_data[self.attack_data['attack_type'] == attack_type]['count']
            report['summary_statistics'][attack_type] = {
                'mean': float(data.mean()),
                'std': float(data.std()),
                'median': float(data.median()),
                'q1': float(data.quantile(0.25)),
                'q3': float(data.quantile(0.75)),
                'max': float(data.max()),
                'total': int(data.sum())
            }

        print("\n" + "*60)
        print("ATTACK PATTERN STATISTICAL ANALYSIS REPORT")
        print("*60)

        print("\nSummary Statistics by Attack Type:")
        print("-*60)

        summary_df = pd.DataFrame(report['summary_statistics']).T
        print(summary_df.round(2))

    return report

# Example usage
analyzer = AttackPatternAnalyzer()

# Generate synthetic data
print("Generating attack data...")
attack_data = analyzer.generate_attack_data(days=30)

print(f"\nDataset shape: {attack_data.shape}")
print(f"Attack types: {attack_data['attack_type'].unique()}")
print(f"Time range: {attack_data['timestamp'].min()} to {attack_data['timestamp'].max()}"
```

```
# Run analyses
print("\n1. Temporal Analysis")
analyzer.temporal_analysis()

print("\n2. Distribution Analysis")
analyzer.distribution_analysis()

print("\n3. Correlation Analysis")
analyzer.correlation_analysis()

print("\n4. Anomaly Detection")
analyzer.anomaly_detection()

print("\n5. Predictive Analysis")
analyzer.predictive_analysis()

# Generate report
report = analyzer.generate_report()
```

---

```

import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix
from collections import defaultdict
import re
import math
from datetime import datetime

class EnhancedBayesianClassifier:
    """
    Advanced Bayesian classifier with multiple enhancements
    """

    def __init__(self, alpha=1.0):
        self.alpha = alpha # Laplace smoothing parameter
        self.class_priors = {}
        self.feature_probs = defaultdict(lambda: defaultdict(dict))
        self.vocabulary = set()
        self.feature_extractors = []
        self.class_counts = {}
        self.total_docs = 0

        # Feature extraction patterns
        self.url_pattern = re.compile(r'https?://[^\s]+')
        self.ip_pattern = re.compile(r'\b(?:[0-9]{1,3}\.){3}[0-9]{1,3}\b')
        self.email_pattern = re.compile(r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-
9.-]+\.[A-Z|a-z]{2,}\b')
        self.script_pattern = re.compile(r'<script[^>]*>.*?</script>',
re.DOTALL | re.IGNORECASE)
        self.sql_pattern =
re.compile(r'\b(SELECT|INSERT|UPDATE|DELETE|DROP|UNION|WHERE)\b',
re.IGNORECASE)

    def extract_features(self, text):
        """Extract multiple types of features from text"""
        features = {}

        # Basic text features
        words = text.lower().split()
        features['word_count'] = len(words)
        features['char_count'] = len(text)
        features['avg_word_length'] = np.mean([len(w) for w in words]) if
words else 0

        # Special character ratios
        features['special_char_ratio'] = len(re.findall(r'[@#$%^&*(),.?":
{}|<>]', text)) / max(len(text), 1)
        features['digit_ratio'] = len(re.findall(r'\d', text)) /

```

```
max(len(text), 1)
    features['uppercase_ratio'] = sum(1 for c in text if c.isupper())
/ max(len(text), 1)

    # Pattern-based features
    features['has_url'] = 1 if self.url_pattern.search(text) else 0
    features['url_count'] = len(self.url_pattern.findall(text))
    features['has_ip'] = 1 if self.ip_pattern.search(text) else 0
    features['has_email'] = 1 if self.email_pattern.search(text) else
0
        features['has_script'] = 1 if self.script_pattern.search(text)
else 0
    features['has_sql'] = 1 if self.sql_pattern.search(text) else 0

    # Suspicious keywords
    suspicious_keywords = ['free', 'click', 'urgent', 'winner',
'guarantee',
                           'risk-free', 'act now', 'limited time',
'veirus', 'malware']
    features['suspicious_word_count'] = sum(1 for word in words
                                             if any(keyword in word for
keyword in suspicious_keywords))

    # N-gram features
    bigrams = [f"{words[i]}_{words[i+1]}" for i in
range(len(words)-1)]
    trigrams = [f"{words[i]}_{words[i+1]}_{words[i+2]}" for i in
range(len(words)-2)]

    # Add words and n-grams to features
    for word in words:
        features[f'word_{word}'] = 1

    for bigram in bigrams[:20]: # Limit to prevent feature explosion
        features[f'bigram_{bigram}'] = 1

    for trigram in trigrams[:10]:
        features[f'trigram_{trigram}'] = 1

    return features

def fit(self, X, y):
    """Train the Bayesian classifier"""
    self.total_docs = len(X)

    # Calculate class priors
    for label in y:
        self.class_counts[label] = self.class_counts.get(label, 0) + 1

    for label, count in self.class_counts.items():
```

```

        self.class_priors[label] = count / self.total_docs

    # Extract features and calculate probabilities
    for text, label in zip(X, y):
        features = self.extract_features(text)

        for feature, value in features.items():
            self.vocabulary.add(feature)

            if feature not in self.feature_probs[label]:
                self.feature_probs[label][feature] = defaultdict(int)

            if isinstance(value, (int, float)):
                # For numeric features, store distribution parameters
                if f'{feature}_values' not in
self.feature_probs[label]:
                    self.feature_probs[label][f'{feature}_values'] =
[]
                    self.feature_probs[label]
[f'{feature}_values'].append(value)
                else:
                    # For categorical features, count occurrences
                    self.feature_probs[label][feature][value] += 1

    # Calculate probability distributions
    for label in self.class_counts:
        for feature in self.vocabulary:
            if f'{feature}_values' in self.feature_probs[label]:
                # Calculate mean and std for numeric features
                values = self.feature_probs[label]
[f'{feature}_values']
                self.feature_probs[label][feature] = {
                    'mean': np.mean(values),
                    'std': np.std(values) + 1e-6 # Avoid division by
zero
                }
            elif feature in self.feature_probs[label]:
                # Calculate probabilities for categorical features
                total = sum(self.feature_probs[label]
[feature].values())
                for value in self.feature_probs[label][feature]:
                    self.feature_probs[label][feature][value] = (
                        self.feature_probs[label][feature][value] +
self.alpha) /
                        (total + self.alpha *
len(self.feature_probs[label][feature])))
                )

    return self

```

```

def predict_proba(self, X):
    """Predict class probabilities"""
    probabilities = []

    for text in X:
        features = self.extract_features(text)
        class_probs = {}

        for label in self.class_priors:
            # Start with class prior (log probability)
            log_prob = math.log(self.class_priors[label])

            for feature, value in features.items():
                if feature in self.feature_probs[label]:
                    if isinstance(self.feature_probs[label][feature], dict) and 'mean' in self.feature_probs[label][feature]:
                        # Gaussian probability for numeric features
                        mean = self.feature_probs[label][feature]['mean']
                        std = self.feature_probs[label][feature]['std']
                        prob = self._gaussian_probability(value, mean, std)
                        log_prob += math.log(prob) if prob > 0 else -10 # Avoid log(0)
                    else:
                        # Categorical probability
                        if value in self.feature_probs[label][feature]:
                            prob = self.feature_probs[label][feature][value]
                        else:
                            # Unseen value - use smoothing
                            prob = self.alpha / (self.class_counts[label] + self.alpha * 2)
                            log_prob += math.log(prob) if prob > 0 else -10
                        else:
                            # Unseen feature - use smoothing
                            prob = self.alpha / (self.class_counts[label] + self.alpha * len(self.vocabulary))
                            log_prob += math.log(prob) if prob > 0 else -10
                class_probs[label] = log_prob

            # Convert log probabilities to probabilities
            max_log_prob = max(class_probs.values())
            exp_probs = {label: math.exp(log_prob - max_log_prob) for label, log_prob in class_probs.items()}
            total_prob = sum(exp_probs.values())

```

```

        normalized_probs = {label: prob/total_prob for label, prob in
exp_probs.items()}
        probabilities.append(normalized_probs)

    return probabilities

def predict(self, X):
    """Predict class labels"""
    probabilities = self.predict_proba(X)
    predictions = []

    for probs in probabilities:
        predictions.append(max(probs, key=probs.get))

    return predictions

def _gaussian_probability(self, x, mean, std):
    """Calculate Gaussian probability"""
    exponent = math.exp(-((x - mean) ** 2) / (2 * std ** 2))
    return (1 / (math.sqrt(2 * math.pi) * std)) * exponent

def explain_prediction(self, text):
    """Explain why a prediction was made"""
    features = self.extract_features(text)
    probabilities = self.predict_proba([text])[0]
    prediction = max(probabilities, key=probabilities.get)

    explanation = {
        'prediction': prediction,
        'probabilities': probabilities,
        'important_features': {}
    }

    # Find most influential features
    for label in self.class_priors:
        feature_impacts = []

        for feature, value in features.items():
            if feature in self.feature_probs[label]:
                if isinstance(self.feature_probs[label][feature], dict) and 'mean' in self.feature_probs[label][feature]:
                    mean = self.feature_probs[label][feature]['mean']
                    impact = abs(value - mean) /
self.feature_probs[label][feature]['std']
                else:
                    if value in self.feature_probs[label][feature]:
                        impact = self.feature_probs[label][feature]
[value]
                    else:

```

```
        impact = 0

        feature_impacts.append((feature, value, impact))

    # Sort by impact
    feature_impacts.sort(key=lambda x: x[2], reverse=True)
    explanation['important_features'][label] =
feature_impacts[:10]

    return explanation

# Example usage with security-related classification
def generate_security_dataset():
    """Generate dataset for security threat classification"""
    data = []

    # Spam emails
    spam_templates = [
        "Congratulations! You've won ${amount}. Click here to claim your
prize!",
        "URGENT: Your account will be suspended. Verify now at {url}",
        "Get rich quick! Make ${amount} working from home. Limited time
offer!",
        "Hot singles in your area! Click here to meet them now!",
        "Discount pharmacy - buy {drug} without prescription. Best prices
guaranteed!"
    ]

    # Phishing attempts
    phishing_templates = [
        "Dear customer, your {bank} account requires verification. Login
at {url}",
        "Security alert: Suspicious activity on your account. Reset
password at {url}",
        "Your {service} subscription is expiring. Update payment method at
{url}",
        "IRS Notice: You have a tax refund of ${amount}. Claim at {url}",
        "IT Department: Your email quota is full. Increase storage at
{url}"
    ]

    # SQL injection attempts
    sql_injection_templates = [
        "' OR '1'='1' --",
        "admin'; DROP TABLE users; --",
        "1' UNION SELECT username, password FROM users --",
        "' OR 1=1; DELETE FROM products WHERE '1'='1",
        "' ; INSERT INTO admin_users VALUES ('hacker', 'password'); --"
    ]
```

```

# Legitimate messages
legitimate_templates = [
    "Meeting scheduled for tomorrow at 2 PM in conference room B",
    "Please review the attached quarterly report and provide
feedback",
    "Reminder: Team lunch on Friday at 12:30 PM",
    "Project update: We've completed 75% of the development phase",
    "Thank you for your recent purchase. Your order #12345 has
shipped"
]

# Generate samples
for _ in range(200):
    # Spam
    template = np.random.choice(spam_templates)
    text = template.format(
        amount=np.random.randint(1000, 10000),
        url=f"http://suspicious-{np.random.randint(1, 100)}.com",
        drug=np.random.choice(['viagra', 'cialis', 'xanax']))
    )
    data.append((text, 'spam'))

    # Phishing
    template = np.random.choice(phishing_templates)
    text = template.format(
        bank=np.random.choice(['Chase', 'Bank of America', 'Wells
Fargo']),
        service=np.random.choice(['Netflix', 'Amazon', 'PayPal']),
        amount=np.random.randint(100, 1000),
        url=f"http://phishing-{np.random.randint(1, 100)}.com"
    )
    data.append((text, 'phishing'))

    # SQL Injection
    text = np.random.choice(sql_injection_templates)
    if np.random.random() > 0.5:
        # Add some context
        text = f"Username: admin Password: {text}"
    data.append((text, 'sql_injection'))

    # Legitimate
    text = np.random.choice(legitimate_templates)
    data.append((text, 'legitimate'))

# Convert to DataFrame
df = pd.DataFrame(data, columns=['text', 'label'])
return df.sample(frac=1).reset_index(drop=True) # Shuffle

# Generate dataset
print("Generating security dataset...")

```

```
df = generate_security_dataset()
print(f"Dataset shape: {df.shape}")
print(f"Class distribution:\n{df['label'].value_counts()}")

# Split data
X_train, X_test, y_train, y_test = train_test_split(
    df['text'].values, df['label'].values, test_size=0.3, random_state=42
)

# Train classifier
print("\nTraining Enhanced Bayesian Classifier...")
classifier = EnhancedBayesianClassifier(alpha=1.0)
classifier.fit(X_train, y_train)

# Make predictions
print("\nMaking predictions...")
y_pred = classifier.predict(X_test)

# Evaluate
print("\nClassification Report:")
print(classification_report(y_test, y_pred))

# Confusion Matrix
print("\nConfusion Matrix:")
cm = confusion_matrix(y_test, y_pred)
labels = sorted(df['label'].unique())
cm_df = pd.DataFrame(cm, index=labels, columns=labels)
print(cm_df)

# Example explanations
print("\n" + "="*60)
print("PREDICTION EXPLANATIONS")
print("=*60)

test_samples = [
    "URGENT: Your PayPal account has been limited. Verify at
http://paypal-secure.fake.com",
    "Meeting rescheduled to 3 PM tomorrow",
    "' OR 1=1; DROP TABLE users; --"
]

for i, sample in enumerate(test_samples):
    print(f"\nSample {i+1}: {sample[:50]}...")
    explanation = classifier.explain_prediction(sample)
    print(f"Prediction: {explanation['prediction']}")
    print(f"Probabilities: {', '.join(f'{k}: {v:.3f}' for k, v in
explanation['probabilities'].items())}")

    print("\nTop features contributing to prediction:")
    for feature, value, impact in explanation['important_features']
```

```
[explanation['prediction']][:5]:  
    if isinstance(value, float):  
        print(f" - {feature}: {value:.3f} ({impact: .3f})")  
    else:  
        print(f" - {feature}: {value} ({impact: .3f})")
```

---

## Exercise 1: Multi-Algorithm Comparison

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.cluster import KMeans, DBSCAN, AgglomerativeClustering
from sklearn.mixture import GaussianMixture
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.metrics import silhouette_score, calinski_harabasz_score,
davies_bouldin_score
from sklearn.model_selection import train_test_split
import warnings
warnings.filterwarnings('ignore')

class MultiAlgorithmAnomalyDetector:
    """
    Compare multiple clustering algorithms for anomaly detection
    """

    def __init__(self):
        self.algorithms = {}
        self.results = {}
        self.scaler = StandardScaler()
        self.best_algorithm = None

    def generate_security_data(self, n_samples=2000):
        """Generate complex security data with labeled anomalies"""
        np.random.seed(42)

        # Normal network behavior (multiple patterns)
        n_normal = int(n_samples * 0.85)
        normal_patterns = []

        # Pattern 1: Regular office hours traffic
        pattern1_samples = n_normal // 3
        pattern1 = np.random.multivariate_normal(
            mean=[50, 1000, 0.1, 20], # connections, bytes, error_rate,
duration
            cov=[[100, 200, 0.01, 5],
                  [200, 10000, 0.05, 10],
                  [0.01, 0.05, 0.001, 0.01],
                  [5, 10, 0.01, 25]],
            size=pattern1_samples
        )

        # Pattern 2: Automated system traffic
        pattern2_samples = n_normal // 3
```

```

pattern2 = np.random.multivariate_normal(
    mean=[200, 500, 0.05, 5],
    cov=[[400, 100, 0.01, 1],
        [100, 2500, 0.01, 2],
        [0.01, 0.01, 0.0001, 0.001],
        [1, 2, 0.001, 4]],
    size=pattern2_samples
)

# Pattern 3: Backup/maintenance traffic
pattern3_samples = n_normal - pattern1_samples - pattern2_samples
pattern3 = np.random.multivariate_normal(
    mean=[30, 50000, 0.01, 300],
    cov=[[25, 5000, 0.001, 50],
        [5000, 5000000, 0.1, 500],
        [0.001, 0.1, 0.0001, 0.01],
        [50, 500, 0.01, 10000]],
    size=pattern3_samples
)

normal_data = np.vstack([pattern1, pattern2, pattern3])

# Anomalous patterns
n_anomalies = n_samples - n_normal
anomalies = []

# Type 1: Port scanning
scan_anomalies = np.random.multivariate_normal(
    mean=[1000, 100, 0.8, 1],
    cov=[[10000, 10, 0.1, 0.1],
        [10, 100, 0.01, 0.1],
        [0.1, 0.01, 0.01, 0.001],
        [0.1, 0.1, 0.001, 0.1]],
    size=n_anomalies // 3
)

# Type 2: Data exfiltration
exfil_anomalies = np.random.multivariate_normal(
    mean=[10, 1000000, 0.02, 1000],
    cov=[[4, 100000, 0.001, 100],
        [100000, 100000000, 0.1, 1000],
        [0.001, 0.1, 0.0001, 0.1],
        [100, 1000, 0.1, 50000]],
    size=n_anomalies // 3
)

# Type 3: DDoS
ddos_anomalies = np.random.multivariate_normal(
    mean=[5000, 1000, 0.9, 0.5],
    cov=[[100000, 100, 0.01, 0.01],

```

```

        [100, 1000, 0.01, 0.01],
        [0.01, 0.01, 0.001, 0.001],
        [0.01, 0.01, 0.001, 0.01]],
        size=n_anomalies - 2 * (n_anomalies // 3)
    )

anomaly_data = np.vstack([scan_anomalies, exfil_anomalies,
ddos_anomalies])

# Combine and create labels
X = np.vstack([normal_data, anomaly_data])
y = np.array([0] * len(normal_data) + [1] * len(anomaly_data))

# Ensure positive values
X = np.abs(X)

# Shuffle
indices = np.random.permutation(len(X))
X, y = X[indices], y[indices]

# Create DataFrame with feature names
feature_names = ['connections_per_min', 'bytes_transferred',
'error_rate', 'avg_duration']
df = pd.DataFrame(X, columns=feature_names)
df['is_anomaly'] = y
342

return df

def apply_kmeans(self, X, k_range=range(2, 10)):
    """Apply K-Means with optimal K selection"""
    print("\n==== K-Means Clustering ===")

    scores = {
        'silhouette': [],
        'calinski': [],
        'davies_bouldin': []
    }

    for k in k_range:
        kmeans = KMeans(n_clusters=k, n_init=10, random_state=42)
        labels = kmeans.fit_predict(X)

        scores['silhouette'].append(silhouette_score(X, labels))
        scores['calinski'].append(calinski_harabasz_score(X, labels))
        scores['davies_bouldin'].append(davies_bouldin_score(X,
labels))

    # Find optimal K (highest silhouette score)
    optimal_k = k_range[np.argmax(scores['silhouette'])]
    print(f"Optimal K: {optimal_k}")

```

```

# Train with optimal K
kmeans = KMeans(n_clusters=optimal_k, n_init=10, random_state=42)
labels = kmeans.fit_predict(X)

self.algorithms['kmeans'] = {
    'model': kmeans,
    'labels': labels,
    'params': {'n_clusters': optimal_k},
    'scores': {
        'silhouette': silhouette_score(X, labels),
        'calinski': calinski_harabasz_score(X, labels),
        'davies_bouldin': davies_bouldin_score(X, labels)
    }
}

return labels, scores

def apply_dbscan(self, X):
    """Apply DBSCAN with parameter optimization"""
    print("\n==== DBSCAN Clustering ====")

    # Find optimal parameters using k-distance graph
    from sklearn.neighbors import NearestNeighbors

    k = 5 # MinPts
    nbrs = NearestNeighbors(n_neighbors=k).fit(X)
    distances, indices = nbrs.kneighbors(X)
    k_distances = distances[:, k-1]
    k_distances = np.sort(k_distances)

    # Find elbow point (simplified)
    elbow_idx = int(len(k_distances) * 0.95)
    eps = k_distances[elbow_idx]

    print(f"Selected eps: {eps:.3f}")

    # Try different MinPts values
    best_score = -1
    best_params = {}

    for min_samples in [3, 5, 7, 10]:
        dbscan = DBSCAN(eps=eps, min_samples=min_samples)
        labels = dbscan.fit_predict(X)

        # Skip if only noise
        if len(set(labels)) <= 1:
            continue

        # Calculate score (excluding noise points)

```

```
mask = labels != -1
if mask.sum() > 0:
    score = silhouette_score(X[mask], labels[mask])
    if score > best_score:
        best_score = score
        best_params = {'eps': eps, 'min_samples': min_samples}
        best_labels = labels

print(f"Best parameters: {best_params}")

self.algorithms['dbSCAN'] = {
    'model': DBSCAN(**best_params),
    'labels': best_labels,
    'params': best_params,
    'scores': {
        'silhouette': best_score,
        'n_clusters': len(set(best_labels)) - (1 if -1 in
best_labels else 0),
        'n_noise': list(best_labels).count(-1)
    }
}

return best_labels

def apply_gmm(self, X, n_components_range=range(2, 10)):
    """Apply Gaussian Mixture Model"""
    print("\n==== Gaussian Mixture Model ====")

    # Find optimal number of components using BIC
    bic_scores = []
    aic_scores = []

    for n in n_components_range:
        gmm = GaussianMixture(n_components=n, random_state=42)
        gmm.fit(X)
        bic_scores.append(gmm.bic(X))
        aic_scores.append(gmm.aic(X))

    # Optimal is minimum BIC
    optimal_n = n_components_range[np.argmin(bic_scores)]
    print(f"Optimal components: {optimal_n}")

    # Train with optimal components
    gmm = GaussianMixture(n_components=optimal_n, random_state=42)
    labels = gmm.fit_predict(X)

    # Calculate anomaly scores (negative log-likelihood)
    scores = -gmm.score_samples(X)

    self.algorithms['gmm'] = {
```

```
'model': gmm,
'labels': labels,
'anomaly_scores': scores,
'params': {'n_components': optimal_n},
'scores': {
    'silhouette': silhouette_score(X, labels),
    'bic': gmm.bic(X),
    'aic': gmm.aic(X)
}
}

return labels, scores

def evaluate_anomaly_detection(self, true_labels):
    """Evaluate how well each algorithm separates anomalies"""
    print("\n==== Anomaly Detection Evaluation ====")

    results = {}

    for name, algo_data in self.algorithms.items():
        labels = algo_data['labels']

        if name == 'gmm':
            # Use anomaly scores for GMM
            threshold = np.percentile(algo_data['anomaly_scores'], 85)
            predicted_anomalies = (algo_data['anomaly_scores'] >
threshold).astype(int)
        elif name == ' dbscan':
            # Noise points are anomalies
            predicted_anomalies = (labels == -1).astype(int)
        else:
            # For K-means: smallest cluster or outlier detection
            cluster_sizes = pd.Series(labels).value_counts()
            anomaly_clusters = cluster_sizes[cluster_sizes <
len(labels) * 0.1].index
            predicted_anomalies = np.isin(labels,
anomaly_clusters).astype(int)

        # Calculate metrics
        from sklearn.metrics import precision_score, recall_score,
f1_score

        results[name] = {
            'precision': precision_score(true_labels,
predicted_anomalies),
            'recall': recall_score(true_labels, predicted_anomalies),
            'f1': f1_score(true_labels, predicted_anomalies),
            'accuracy': np.mean(predicted_anomalies == true_labels)
        }

    return results
```

```

        print(f"\n{name.upper()}:")
        for metric, value in results[name].items():
            print(f"  {metric}: {value:.3f}")

    self.results = results
    return results

def visualize_results(self, X, y, feature_names):
    """Visualize clustering results using PCA"""
    # Reduce to 2D for visualization
    pca = PCA(n_components=2)
    X_pca = pca.fit_transform(X)

    fig, axes = plt.subplots(2, 3, figsize=(18, 12))
    axes = axes.flatten()

    # Plot each algorithm's results
    for idx, (name, algo_data) in enumerate(self.algorithms.items()):
        ax = axes[idx]
        labels = algo_data['labels']

        # Create color map
        unique_labels = np.unique(labels)
        colors = plt.cm.viridis(np.linspace(0, 1, len(unique_labels)))

        for i, label in enumerate(unique_labels):
            if label == -1: # Noise in DBSCAN
                color = 'red'
                marker = 'x'
            else:
                color = colors[i]
                marker = 'o'

            mask = labels == label
            ax.scatter(X_pca[mask, 0], X_pca[mask, 1],
                       c=[color], marker=marker, alpha=0.6,
                       label=f'Cluster {label}' if label != -1 else
                           'Noise')

    # Highlight true anomalies
    anomaly_mask = y == 1
    ax.scatter(X_pca[anomaly_mask, 0], X_pca[anomaly_mask, 1],
               facecolors='none', edgecolors='red', s=100,
               linewidths=2,
               label='True Anomalies')

    ax.set_title(f'{name.upper()}\nSilhouette:\n{algo_data["scores"]["silhouette", "N/A"]:.3f}')
    ax.set_xlabel(f'PC1 ({pca.explained_variance_ratio_[0]:.2%})')
    ax.set_ylabel(f'PC2 ({pca.explained_variance_ratio_[1]:.2%})')

```

```

if idx == 0:
    ax.legend(bbox_to_anchor=(1.05, 1), loc='upper left')

# Ground truth
ax = axes[3]
scatter = ax.scatter(X_pca[:, 0], X_pca[:, 1], c=y, cmap='RdBu',
alpha=0.6)
ax.set_title('Ground Truth')
ax.set_xlabel(f'PC1 ({pca.explained_variance_ratio_[0]:.2%})')
ax.set_ylabel(f'PC2 ({pca.explained_variance_ratio_[1]:.2%})')
plt.colorbar(scatter, ax=ax, label='Anomaly')

# Performance comparison
ax = axes[4]
metrics_df = pd.DataFrame(self.results).T
metrics_df.plot(kind='bar', ax=ax)
ax.set_title('Algorithm Performance Comparison')
ax.set_xlabel('Algorithm')
ax.set_ylabel('Score')
ax.legend(loc='lower right')
ax.grid(True, alpha=0.3)

# Recommendations
ax = axes[5]
ax.axis('off')

# Generate recommendations
best_f1 = max(self.results.items(), key=lambda x: x[1]['f1'])
best_precision = max(self.results.items(), key=lambda x: x[1]
['precision'])
best_recall = max(self.results.items(), key=lambda x: x[1]
['recall'])

recommendations = f"""
RECOMMENDATIONS
=====
Best Overall (F1): {best_f1[0].upper()} (F1={best_f1[1]
['f1']:.3f})
    Best Precision: {best_precision[0].upper()} ({best_precision[1]
['precision']:.3f})
    Best Recall: {best_recall[0].upper()} ({best_recall[1]
['recall']:.3f})

Use Cases:
- High precision needed: {best_precision[0].upper()}
    (minimize false positives)
- High recall needed: {best_recall[0].upper()}
    (catch all anomalies)
"""

```

```
- Balanced approach: {best_f1[0].upper()}

    Data Characteristics:
    - {'Dense clusters' if self.algorithms['kmeans']['params']['n_clusters'] <= 5 else 'Many patterns'}
    - {'Contains noise' if self.algorithms['dbSCAN']['scores']['n_noise'] > 0 else 'Clean data'}
    - {'Well-separated' if best_f1[1]['f1'] > 0.7 else 'Overlapping patterns'}
    """

    ax.text(0.1, 0.5, recommendations, transform=ax.transAxes,
            fontsize=11, verticalalignment='center',
            fontfamily='monospace')

    plt.tight_layout()
    plt.show()

def generate_report(self):
    """Generate comprehensive comparison report"""
    report = {
        'data_characteristics': {},
        'algorithm_performance': self.results,
        'recommendations': {},
        'best_algorithm': None
    }

    # Find best algorithm
    best_algo = max(self.results.items(), key=lambda x: x[1]['f1'])
    report['best_algorithm'] = best_algo[0]

    # Generate recommendations based on use case
    report['recommendations'] = {
        'minimize_false_positives': max(self.results.items(),
                                         key=lambda x: x[1]['precision'])[0],
        'catch_all_anomalies': max(self.results.items(),
                                   key=lambda x: x[1]['recall'])[0],
        'balanced_approach': best_algo[0]
    }

    return report

# Example usage
detector = MultiAlgorithmAnomalyDetector()

# Generate data
print("Generating security dataset with anomalies...")
df = detector.generate_security_data(n_samples=1500)
X = df.drop('is_anomaly', axis=1).values
```

```
y = df['is_anomaly'].values

print(f"Dataset shape: {X.shape}")
print(f"Anomaly rate: {y.mean():.2%}")

# Scale data
X_scaled = detector.scaler.fit_transform(X)

# Apply algorithms
kmeans_labels, kmeans_scores = detector.apply_kmeans(X_scaled)
dbscan_labels = detector.apply_dbscan(X_scaled)
gmm_labels, gmm_scores = detector.apply_gmm(X_scaled)

# Evaluate
results = detector.evaluate_anomaly_detection(y)

# Visualize
detector.visualize_results(X_scaled, y, df.columns[:-1])

# Generate report
report = detector.generate_report()
print("\n" + "="*60)
print("FINAL RECOMMENDATION")
print("=*60")
print(f"Best algorithm for this dataset:")
{report['best_algorithm'].upper()}")
print(f"\nUse case recommendations:")
for use_case, algo in report['recommendations'].items():
    print(f" - {use_case.replace('_', ' ').title(): {algo.upper()}}")
```

```

import numpy as np
import pandas as pd
from sklearn.ensemble import IsolationForest
from sklearn.svm import OneClassSVM
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import precision_recall_curve, auc
from collections import deque
import pickle
from datetime import datetime, timedelta
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

class AdaptiveAnomalyDetectionPipeline:
    """
        Adaptive anomaly detection system with multiple algorithms and online
    learning
    """
    def __init__(self, window_size=1000, update_frequency=100):
        self.window_size = window_size
        self.update_frequency = update_frequency

        # Initialize models
        self.models = {
            'isolation_forest': IsolationForest(contamination=0.1,
random_state=42),
            'one_class_svm': OneClassSVM(nu=0.1, kernel='rbf',
gamma='auto'),
            'autoencoder': None # Will be built based on input shape
        }

        # Ensemble weights (will be adapted based on performance)
        self.model_weights = {
            'isolation_forest': 1.0,
            'one_class_svm': 1.0,
            'autoencoder': 1.0
        }

        # Performance tracking
        self.performance_history = {
            model: deque(maxlen=100) for model in self.models
        }

        # Data buffer for adaptation
        self.data_buffer = deque(maxlen=window_size)
        self.feedback_buffer = deque(maxlen=window_size)

        # Threshold adaptation

```

```

        self.thresholds = {
            model: 0.5 for model in self.models
        }

        # Drift detection
        self.baseline_statistics = None
        self.drift_detector = DriftDetector()

        # Scaler
        self.scaler = StandardScaler()
        self.is_fitted = False

    def build_autoencoder(self, input_dim):
        """Build adaptive autoencoder"""
        # Encoder
        inputs = layers.Input(shape=(input_dim,))
        encoded = layers.Dense(max(input_dim // 2, 16), activation='relu')(inputs)
        encoded = layers.BatchNormalization()(encoded)
        encoded = layers.Dense(max(input_dim // 4, 8), activation='relu')(encoded)

        # Decoder
        decoded = layers.Dense(max(input_dim // 2, 16), activation='relu')(encoded)
        decoded = layers.BatchNormalization()(decoded)
        decoded = layers.Dense(input_dim, activation='sigmoid')(decoded)

        # Model
        autoencoder = keras.Model(inputs, decoded)
        autoencoder.compile(optimizer='adam', loss='mse')

        return autoencoder

    def initial_fit(self, X_normal):
        """Initial training on normal data"""
        print("Initial model fitting...")

        # Fit scaler
        self.scaler.fit(X_normal)
        X_scaled = self.scaler.transform(X_normal)

        # Fit traditional models
        self.models['isolation_forest'].fit(X_scaled)
        self.models['one_class_svm'].fit(X_scaled)

        # Build and train autoencoder
        self.models['autoencoder'] =
self.build_autoencoder(X_scaled.shape[1])
        self.models['autoencoder'].fit(

```

```
X_scaled, X_scaled,
epochs=50,
batch_size=32,
validation_split=0.1,
verbose=0
)

# Calculate initial thresholds
self._calculate_thresholds(X_scaled)

# Store baseline statistics
self.baseline_statistics = {
    'mean': X_scaled.mean(axis=0),
    'std': X_scaled.std(axis=0),
    'min': X_scaled.min(axis=0),
    'max': X_scaled.max(axis=0)
}

self.is_fitted = True
print("Initial fitting complete!")

def _calculate_thresholds(self, X):
    """Calculate anomaly thresholds for each model"""
    # Isolation Forest
    if_scores = self.models['isolation_forest'].score_samples(X)
    self.thresholds['isolation_forest'] = np.percentile(if_scores, 5)

    # One-Class SVM
    svm_scores = self.models['one_class_svm'].score_samples(X)
    self.thresholds['one_class_svm'] = np.percentile(svm_scores, 5)

    # Autoencoder
    reconstructions = self.models['autoencoder'].predict(X, verbose=0)
    mse = np.mean((X - reconstructions) ** 2, axis=1)
    self.thresholds['autoencoder'] = np.percentile(mse, 95)

def predict_single(self, x):
    """Predict anomaly for single instance"""
    if not self.is_fitted:
        raise ValueError("Model not fitted yet!")

    x_scaled = self.scaler.transform(x.reshape(1, -1))

    scores = {}
    predictions = {}

    # Isolation Forest
    if_score = self.models['isolation_forest'].score_samples(x_scaled)
[0]
    scores['isolation_forest'] = if_score
```

```

        predictions['isolation_forest'] = int(if_score <
self.thresholds['isolation_forest'])

        # One-Class SVM
        svm_score = self.models['one_class_svm'].score_samples(x_scaled)
[0]
        scores['one_class_svm'] = svm_score
        predictions['one_class_svm'] = int(svm_score <
self.thresholds['one_class_svm'])

        # Autoencoder
        reconstruction = self.models['autoencoder'].predict(x_scaled,
verbose=0)
        ae_score = np.mean((x_scaled - reconstruction) ** 2)
        scores['autoencoder'] = ae_score
        predictions['autoencoder'] = int(ae_score >
self.thresholds['autoencoder'])

        # Ensemble prediction
        weighted_sum = sum(
            predictions[model] * self.model_weights[model]
            for model in self.models
        )
        total_weight = sum(self.model_weights.values())
        ensemble_score = weighted_sum / total_weight

        return {
            'prediction': int(ensemble_score > 0.5),
            'confidence': abs(ensemble_score - 0.5) * 2,
            'ensemble_score': ensemble_score,
            'individual_predictions': predictions,
            'individual_scores': scores
        }

    def update_with_feedback(self, x, true_label):
        """Update models based on feedback"""
        # Add to buffers
        self.data_buffer.append(x)
        self.feedback_buffer.append(true_label)

        # Get prediction
        result = self.predict_single(x)

        # Update performance metrics
        for model in self.models:
            correct = (result['individual_predictions'][model] ==
true_label)
            self.performance_history[model].append(correct)

        # Adapt weights based on recent performance

```

```

    if len(self.data_buffer) % self.update_frequency == 0:
        self._adapt_weights()

    # Check for concept drift
    if self._detect_drift():
        print("Concept drift detected! Retraining models...")
        self._retrain_models()

    def _adapt_weights(self):
        """Adapt ensemble weights based on recent performance"""
        for model in self.models:
            if len(self.performance_history[model]) > 10:
                recent_accuracy =
                    np.mean(list(self.performance_history[model])[-50:])
                # Update weight (higher accuracy = higher weight)
                self.model_weights[model] = max(0.1, recent_accuracy)

        # Normalize weights
        total_weight = sum(self.model_weights.values())
        for model in self.models:
            self.model_weights[model] /= total_weight

    def _detect_drift(self):
        """Detect concept drift in data"""
        if len(self.data_buffer) < 100:
            return False

```

354

```

        recent_data = np.array(list(self.data_buffer)[-100:])
        recent_scaled = self.scaler.transform(recent_data)

        # Check statistical drift
        drift_detected = False

        for i in range(recent_scaled.shape[1]):
            # Kolmogorov-Smirnov test
            from scipy.stats import ks_2samp

            baseline_dist = self.baseline_statistics['mean'][i]
            recent_dist = recent_scaled[:, i].mean()

            # Simple threshold-based drift detection
            if abs(baseline_dist - recent_dist) > 2 *
                self.baseline_statistics['std'][i]:
                drift_detected = True
                break

        return drift_detected

    def _retrain_models(self):
        """Retrain models on recent data"""

```

```

if len(self.data_buffer) < 100:
    return

# Get recent normal data
recent_data = np.array(list(self.data_buffer)[-500:])
recent_labels = np.array(list(self.feedback_buffer)[-500:])
normal_data = recent_data[recent_labels == 0]

if len(normal_data) < 50:
    return

# Update scaler
self.scaler.fit(normal_data)
normal_scaled = self.scaler.transform(normal_data)

# Retrain models
try:
    self.models['isolation_forest'].fit(normal_scaled)
    self.models['one_class_svm'].fit(normal_scaled)

    # Fine-tune autoencoder
    self.models['autoencoder'].fit(
        normal_scaled, normal_scaled,
        epochs=10,
        batch_size=32,
        verbose=0
    )

    # Recalculate thresholds
    self._calculate_thresholds(normal_scaled)

    # Update baseline statistics
    self.baseline_statistics = {
        'mean': normal_scaled.mean(axis=0),
        'std': normal_scaled.std(axis=0),
        'min': normal_scaled.min(axis=0),
        'max': normal_scaled.max(axis=0)
    }

    print("Models retrained successfully!")

except Exception as e:
    print(f"Error during retraining: {e}")

def get_model_performance(self):
    """Get current model performance metrics"""
    performance = {}

    for model in self.models:
        if len(self.performance_history[model]) > 0:

```

```

        performance[model] = {
            'recent_accuracy': np.mean(list(self.performance_history[model])[-50:]),
            'weight': self.model_weights[model],
            'total_predictions': len(self.performance_history[model])
        }

    return performance

```

## Section 3: Machine Learning Essentials Exercise Solutions (Continued)

### Exercise 2: Adaptive Anomaly Detection (Continued)

```

```python
def save_state(self, filepath):
    """Save pipeline state"""
    state = {
        'scaler': self.scaler,
        'thresholds': self.thresholds,
        'model_weights': self.model_weights,
        'baseline_statistics': self.baseline_statistics,
        'performance_history': dict(self.performance_history),
        'models': {
            'isolation_forest': self.models['isolation_forest'],
            'one_class_svm': self.models['one_class_svm']
        }
    }

    # Save autoencoder separately
    self.models['autoencoder'].save(f"{filepath}_autoencoder.h5")

    with open(filepath, 'wb') as f:
        pickle.dump(state, f)

    print(f"Pipeline state saved to {filepath}")

def load_state(self, filepath):
    """Load pipeline state"""
    with open(filepath, 'rb') as f:
        state = pickle.load(f)

        self.scaler = state['scaler']
        self.thresholds = state['thresholds']
        self.model_weights = state['model_weights']
        self.baseline_statistics = state['baseline_statistics']

    # Restore performance history
    for model, history in state['performance_history'].items():
        self.performance_history[model] = deque(history, maxlen=100)

```

```

# Restore models
self.models['isolation_forest'] = state['models']
['isolation_forest']
self.models['one_class_svm'] = state['models']['one_class_svm']

# Load autoencoder
self.models['autoencoder'] = keras.models.load_model(f"{filepath}_autoencoder.h5")

self.is_fitted = True
print(f"Pipeline state loaded from {filepath}")

class DriftDetector:
    """Helper class for concept drift detection"""
    def __init__(self, window_size=100, threshold=3.0):
        self.window_size = window_size
        self.threshold = threshold
        self.reference_window = deque(maxlen=window_size)
        self.current_window = deque(maxlen=window_size)

    def add_sample(self, error):
        """Add prediction error sample"""
        if len(self.reference_window) < self.window_size:
            self.reference_window.append(error)
        else:
            self.current_window.append(error)

        if len(self.current_window) == self.window_size:
            # Check for drift
            if self.detect_drift():
                # Move current to reference
                self.reference_window = self.current_window.copy()
                self.current_window.clear()
                return True
        return False

    def detect_drift(self):
        """Detect drift using statistical tests"""
        if len(self.current_window) < self.window_size // 2:
            return False

        ref_mean = np.mean(self.reference_window)
        ref_std = np.std(self.reference_window)
        curr_mean = np.mean(self.current_window)

        # Page-Hinkley test
        drift_score = abs(curr_mean - ref_mean) / (ref_std + 1e-6)

```

```

        return drift_score > self.threshold

# Simulation class for testing adaptive behavior
class AnomalyStreamSimulator:
    """Simulate streaming data with concept drift"""
    def __init__(self, n_features=10):
        self.n_features = n_features
        self.current_phase = 0
        self.phase_counter = 0
        self.phases = [
            {'normal_mean': 0, 'normal_std': 1, 'anomaly_mean': 5,
            'anomaly_std': 1},
            {'normal_mean': 2, 'normal_std': 0.5, 'anomaly_mean': -2,
            'anomaly_std': 0.5},
            {'normal_mean': -1, 'normal_std': 2, 'anomaly_mean': 3,
            'anomaly_std': 0.5}
        ]

    def generate_sample(self):
        """Generate next sample"""
        # Change phase periodically
        if self.phase_counter > 500:
            self.current_phase = (self.current_phase + 1) %
len(self.phases)
            self.phase_counter = 0

        self.phase_counter += 1
        phase = self.phases[self.current_phase]

        # 90% normal, 10% anomaly
        is_anomaly = np.random.random() < 0.1

        if is_anomaly:
            sample = np.random.normal(
                phase['anomaly_mean'],
                phase['anomaly_std'],
                self.n_features
            )
        else:
            sample = np.random.normal(
                phase['normal_mean'],
                phase['normal_std'],
                self.n_features
            )

        return sample, int(is_anomaly)

# Example usage

```

```
print("== Adaptive Anomaly Detection Pipeline ==\n")

# Create pipeline
pipeline = AdaptiveAnomalyDetectionPipeline(
    window_size=1000,
    update_frequency=50
)

# Generate initial training data
print("Generating initial training data...")
np.random.seed(42)
X_train = np.random.normal(0, 1, (500, 10))

# Initial fit
pipeline.initial_fit(X_train)

# Create stream simulator
simulator = AnomalyStreamSimulator(n_features=10)

# Simulate streaming detection with adaptation
print("\nSimulating streaming anomaly detection with adaptation...")
results = {
    'predictions': [],
    'true_labels': [],
    'confidences': [],
    'phase_changes': []
}

for i in range(1500):
    # Generate sample
    sample, true_label = simulator.generate_sample()

    # Get prediction
    result = pipeline.predict_single(sample)

    # Store results
    results['predictions'].append(result['prediction'])
    results['true_labels'].append(true_label)
    results['confidences'].append(result['confidence'])

    # Update with feedback
    pipeline.update_with_feedback(sample, true_label)

    # Track phase changes
    if i > 0 and i % 500 == 0:
        results['phase_changes'].append(i)
        print(f"\nPhase change at sample {i}")
        performance = pipeline.get_model_performance()
        print("Current model performance:")
        for model, perf in performance.items():
```

```

        print(f"  {model}: accuracy={perf['recent_accuracy']:.3f},
weight={perf['weight']:.3f}")

# Calculate metrics
from sklearn.metrics import classification_report, accuracy_score

print("\n==== Final Results ===")
print(classification_report(results['true_labels'],
results['predictions']))

# Plot results
import matplotlib.pyplot as plt

fig, axes = plt.subplots(3, 1, figsize=(12, 10))

# Accuracy over time
ax = axes[0]
window = 50
accuracies = []
for i in range(window, len(results['predictions'])):
    acc = accuracy_score(
        results['true_labels'][i-window:i],
        results['predictions'][i-window:i]
    )
    accuracies.append(acc)

ax.plot(range(window, len(results['predictions'])), accuracies)
for phase_change in results['phase_changes']:
    ax.axvline(x=phase_change, color='r', linestyle='--', alpha=0.5)
ax.set_title('Detection Accuracy Over Time (50-sample window)')
ax.set_xlabel('Sample Number')
ax.set_ylabel('Accuracy')
ax.grid(True, alpha=0.3)

# Model weights over time
ax = axes[1]
weight_history = {model: [] for model in pipeline.models}
check_points = range(0, len(results['predictions']), 50)

for checkpoint in check_points:
    if checkpoint > 0:
        # Simulate checking weights at checkpoints
        for model in pipeline.models:
            weight_history[model].append(pipeline.model_weights[model])

for model, weights in weight_history.items():
    ax.plot(check_points[1:], weights, label=model)

ax.set_title('Model Weights Adaptation')
ax.set_xlabel('Sample Number')

```

```
ax.set_ylabel('Weight')
ax.legend()
ax.grid(True, alpha=0.3)

# Confidence distribution
ax = axes[2]
ax.hist(results['confidences'], bins=30, alpha=0.7, edgecolor='black')
ax.set_title('Prediction Confidence Distribution')
ax.set_xlabel('Confidence')
ax.set_ylabel('Frequency')
ax.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

# Save pipeline state
print("\nSaving pipeline state...")
pipeline.save_state('adaptive_pipeline_state.pkl')
```

```

import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler, MinMaxScaler,
RobustScaler
from sklearn.feature_selection import SelectKBest, f_classif,
mutual_info_classif
from sklearn.decomposition import PCA, FastICA
from sklearn.ensemble import RandomForestClassifier
import warnings
warnings.filterwarnings('ignore')

class SecurityFeatureEngineer:
    """
    Comprehensive feature engineering pipeline for security data
    """

    def __init__(self):
        self.feature_generators = []
        self.selected_features = None
        self.transformation_pipeline = None
        self.feature_importance = {}

    def generate_network_features(self, df):
        """Generate network traffic features"""
        print("Generating network traffic features...") 362

        features = pd.DataFrame(index=df.index)

        # Basic statistics
        if 'bytes_sent' in df.columns and 'bytes_received' in df.columns:
            features['bytes_ratio'] = df['bytes_sent'] /
(df['bytes_received'] + 1)
            features['total_bytes'] = df['bytes_sent'] +
df['bytes_received']
            features['bytes_asymmetry'] = (df['bytes_sent'] -
df['bytes_received']) / (features['total_bytes'] + 1)

        # Time-based features
        if 'timestamp' in df.columns:
            df['timestamp'] = pd.to_datetime(df['timestamp'])
            features['hour'] = df['timestamp'].dt.hour
            features['day_of_week'] = df['timestamp'].dt.dayofweek
            features['is_weekend'] = (features['day_of_week'] >=
5).astype(int)
            features['is_business_hours'] = ((features['hour'] >= 9) &
(features['hour'] <= 17)).astype(int)

        # Cyclical encoding
        features['hour_sin'] = np.sin(2 * np.pi * features['hour'] /

```

```

24)
    features['hour_cos'] = np.cos(2 * np.pi * features['hour'] /
24)
    features['dow_sin'] = np.sin(2 * np.pi *
features['day_of_week'] / 7)
    features['dow_cos'] = np.cos(2 * np.pi *
features['day_of_week'] / 7)

    # Connection features
    if 'duration' in df.columns:
        features['duration_log'] = np.log1p(df['duration'])
        features['is_short_connection'] = (df['duration'] <
1).astype(int)
        features['is_long_connection'] = (df['duration'] >
3600).astype(int)

    # Port features
    if 'dest_port' in df.columns:
        features['is_well_known_port'] = (df['dest_port'] <
1024).astype(int)
        features['is_registered_port'] = ((df['dest_port'] >= 1024) &
(df['dest_port'] < 49152)).astype(int)
        features['is_dynamic_port'] = (df['dest_port'] >=
49152).astype(int)

    # Specific service ports
    common_ports = {
        'http': [80, 8080, 8000],
        'https': [443, 8443],
        'ssh': [22],
        'ftp': [20, 21],
        'smtp': [25, 587],
        'dns': [53],
        'sql': [1433, 3306, 5432]
    }

    for service, ports in common_ports.items():
        features[f'is_{service}_port'] =
df['dest_port'].isin(ports).astype(int)

    # Protocol features
    if 'protocol' in df.columns:
        protocol_dummies = pd.get_dummies(df['protocol'],
prefix='protocol')
        features = pd.concat([features, protocol_dummies], axis=1)

    # Flag features
    if 'tcp_flags' in df.columns:
        # Extract individual flags
        flag_names = ['FIN', 'SYN', 'RST', 'PSH', 'ACK', 'URG']

```

```

        for i, flag in enumerate(flag_names):
            features[f'flag_{flag}'] = ((df['tcp_flags'] >> i) &
1).astype(int)

            # Flag combinations
            features['flag_SYN_ACK'] = (features['flag_SYN'] &
features['flag_ACK']).astype(int)
            features['flag_FIN_ACK'] = (features['flag_FIN'] &
features['flag_ACK']).astype(int)

    return features

def generate_statistical_features(self, df, window_sizes=[10, 50,
100]):
    """Generate rolling statistical features"""
    print("Generating statistical features...")

    features = pd.DataFrame(index=df.index)
    numeric_cols = df.select_dtypes(include=[np.number]).columns

    for col in numeric_cols:
        for window in window_sizes:
            # Rolling statistics
            features[f'{col}_rolling_mean_{window}'] =
df[col].rolling(window, min_periods=1).mean()
            features[f'{col}_rolling_std_{window}'] =
df[col].rolling(window, min_periods=1).std()
            features[f'{col}_rolling_min_{window}'] =
df[col].rolling(window, min_periods=1).min()
            features[f'{col}_rolling_max_{window}'] =
df[col].rolling(window, min_periods=1).max()

            # Deviation from rolling mean
            rolling_mean = features[f'{col}_rolling_mean_{window}']
            features[f'{col}_deviation_{window}'] = (df[col] -
rolling_mean) / (rolling_mean + 1e-8)

            # Exponential weighted statistics
            features[f'{col}_ewm_mean'] = df[col].ewm(span=20).mean()
            features[f'{col}_ewm_std'] = df[col].ewm(span=20).std()

    return features

def generate_interaction_features(self, df, max_degree=2):
    """Generate polynomial and interaction features"""
    print("Generating interaction features...")

    features = pd.DataFrame(index=df.index)
    numeric_cols = df.select_dtypes(include=[np.number]).columns[:10]
# Limit to prevent explosion

```

```

# Polynomial features
for col in numeric_cols:
    for degree in range(2, max_degree + 1):
        features[f'{col}_pow_{degree}'] = df[col] ** degree

# Interaction features
for i, col1 in enumerate(numeric_cols):
    for col2 in numeric_cols[i+1:]:
        features[f'{col1}_times_{col2}'] = df[col1] * df[col2]
        features[f'{col1}_div_{col2}'] = df[col1] / (df[col2] +
1e-8)

return features

def generate_frequency_features(self, df, categorical_cols):
    """Generate frequency-based features"""
    print("Generating frequency features...")

features = pd.DataFrame(index=df.index)

for col in categorical_cols:
    if col in df.columns:
        # Value counts
        value_counts = df[col].value_counts()
        features[f'{col}_frequency'] = df[col].map(value_counts)
        features[f'{col}_frequency_ratio'] =
features[f'{col}_frequency'] / len(df)

        # Rare value indicator
        threshold = 0.01 * len(df)
        features[f'{col}_is_rare'] = (features[f'{col}_frequency'] < threshold).astype(int)

return features

def generate_embedding_features(self, df, categorical_cols,
n_components=10):
    """Generate embeddings for categorical variables"""
    print("Generating embedding features...")

features = pd.DataFrame(index=df.index)

for col in categorical_cols:
    if col in df.columns:
        # Target encoding (simplified - in practice use cross-
validation)
        if 'label' in df.columns:
            target_encode = df.groupby(col)['label'].mean()
            features[f'{col}_target_encoded'] =

```

```

df[col].map(target_encode)

        # One-hot encoding with dimensionality reduction
        if df[col].nunique() > n_components:
            dummies = pd.get_dummies(df[col], prefix=col)
            pca = PCA(n_components=n_components)
            embeddings = pca.fit_transform(dummies)

            for i in range(n_components):
                features[f'{col}_embed_{i}'] = embeddings[:, i]

    return features

def select_features(self, X, y, n_features=50, methods=['mutual_info',
'rf_importance']):
    """Select most important features"""
    print(f"\nSelecting top {n_features} features...")

    feature_scores = {}

    # Mutual Information
    if 'mutual_info' in methods:
        mi_scores = mutual_info_classif(X, y)
        feature_scores['mutual_info'] = dict(zip(X.columns,
mi_scores))

    # Random Forest Importance
    if 'rf_importance' in methods:
        rf = RandomForestClassifier(n_estimators=100, random_state=42)
        rf.fit(X, y)
        feature_scores['rf_importance'] = dict(zip(X.columns,
rf.feature_importances_))

    # ANOVA F-value
    if 'f_value' in methods:
        f_scores = f_classif(X, y)[0]
        feature_scores['f_value'] = dict(zip(X.columns, f_scores))

    # Combine scores
    combined_scores = {}
    for feature in X.columns:
        scores = [feature_scores[method].get(feature, 0) for method in
methods]
        # Normalize scores
        normalized_scores = []
        for i, method in enumerate(methods):
            all_scores = list(feature_scores[method].values())
            max_score = max(all_scores) if all_scores else 1
            normalized_scores.append(scores[i] / max_score)
        combined_scores[feature] = np.mean(normalized_scores)

```

```

        # Select top features
        sorted_features = sorted(combined_scores.items(), key=lambda x:
x[1], reverse=True)
        self.selected_features = [f[0] for f in
sorted_features[:n_features]]
        self.feature_importance = dict(sorted_features[:n_features])

    return self.selected_features

def create_feature_pipeline(self, df, target_col='label'):
    """Create complete feature engineering pipeline"""
    print("\n==== Creating Feature Engineering Pipeline ====")

    # Separate features and target
    if target_col in df.columns:
        y = df[target_col]
        df_features = df.drop(columns=[target_col])
    else:
        y = None
        df_features = df

    # Identify column types
    numeric_cols = df_features.select_dtypes(include=
[np.number]).columns.tolist()
    categorical_cols = df_features.select_dtypes(include=
['object']).columns.tolist()

    # Generate all features
    all_features = []

    # 1. Network features
    network_features = self.generate_network_features(df_features)
    all_features.append(network_features)

    # 2. Statistical features
    if numeric_cols:
        stat_features =
self.generate_statistical_features(df_features[numeric_cols])
        all_features.append(stat_features)

    # 3. Interaction features
    if len(numeric_cols) > 1:
        interaction_features =
self.generate_interaction_features(df_features[numeric_cols[:5]])
        all_features.append(interaction_features)

    # 4. Frequency features
    if categorical_cols:
        freq_features = self.generate_frequency_features(df_features,

```

```

categorical_cols)
    all_features.append(freq_features)

    # 5. Embedding features
    if categorical_cols and y is not None:
        df_with_label = df_features.copy()
        df_with_label['label'] = y
        embed_features =
self.generate_embedding_features(df_with_label, categorical_cols[:3])
        all_features.append(embed_features)

    # Combine all features
    feature_matrix = pd.concat(all_features, axis=1)

    # Add original numeric features
    feature_matrix = pd.concat([df_features[numeric_cols],
feature_matrix], axis=1)

    # Handle missing values
    feature_matrix = feature_matrix.fillna(0)

    print(f"\nGenerated {feature_matrix.shape[1]} features")

    # Feature selection if target is available
    if y is not None:
        selected = self.select_features(feature_matrix, y,
n_features=50)
        feature_matrix = feature_matrix[selected]
        print(f"Selected {len(selected)} features")

    return feature_matrix

def visualize_feature_importance(self, top_n=20):
    """Visualize feature importance"""
    if not self.feature_importance:
        print("No feature importance data available")
        return

    import matplotlib.pyplot as plt

    # Get top features
    top_features = list(self.feature_importance.items())[:top_n]
    features, scores = zip(*top_features)

    # Create plot
    plt.figure(figsize=(10, 8))
    y_pos = np.arange(len(features))

    plt.barh(y_pos, scores)
    plt.yticks(y_pos, features)

```

```

        plt.xlabel('Importance Score')
        plt.title(f'Top {top_n} Feature Importance')
        plt.tight_layout()
        plt.show()

    def generate_feature_report(self):
        """Generate feature engineering report"""
        report = {
            'total_features_generated': len(self.feature_importance) if self.feature_importance else 0,
            'selected_features': self.selected_features,
            'top_10_features': list(self.feature_importance.items())[:10]
        }
        if self.feature_importance else [],
            'feature_categories': {
                'network': sum(1 for f in self.selected_features if any(x in f for x in ['bytes', 'port', 'protocol', 'flag'])),
                'temporal': sum(1 for f in self.selected_features if any(x in f for x in ['hour', 'day', 'time'])),
                'statistical': sum(1 for f in self.selected_features if any(x in f for x in ['rolling', 'ewm', 'std', 'mean'])),
                'interaction': sum(1 for f in self.selected_features if any(x in f for x in ['times', 'div', 'pow'])),
            } if self.selected_features else {}
        }

        return report

```

369

```

# Example usage
print("== Security Feature Engineering Pipeline ==\n")

# Generate sample security data
np.random.seed(42)
n_samples = 1000

# Create synthetic network traffic data
data = {
    'timestamp': pd.date_range(start='2024-01-01', periods=n_samples,
                                freq='1min'),
    'bytes_sent': np.random.lognormal(10, 2, n_samples),
    'bytes_received': np.random.lognormal(10, 2, n_samples),
    'duration': np.random.exponential(30, n_samples),
    'dest_port': np.random.choice([80, 443, 22, 3306, 8080, 53, 25,
                                    np.random.randint(49152, 65535)],
                                n_samples,
                                p=[0.3, 0.3, 0.1, 0.05, 0.05, 0.05, 0.05,
                                0.1]),
    'protocol': np.random.choice(['TCP', 'UDP', 'ICMP'], n_samples, p=
[0.7, 0.25, 0.05]),
    'tcp_flags': np.random.randint(0, 64, n_samples),
}

```

```

'src_ip': np.random.choice([f'192.168.1.{i}' for i in range(1, 20)],
n_samples),
'label': np.random.choice([0, 1], n_samples, p=[0.9, 0.1]) # 10%
anomalies
}

df = pd.DataFrame(data)

# Initialize feature engineer
engineer = SecurityFeatureEngineer()

# Create feature pipeline
feature_matrix = engineer.create_feature_pipeline(df, target_col='label')

print(f"\nFinal feature matrix shape: {feature_matrix.shape}")
print(f"Features created: {list(feature_matrix.columns[:10])}...")

# Visualize feature importance
engineer.visualize_feature_importance(top_n=20)

# Generate report
report = engineer.generate_feature_report()
print("\n==== Feature Engineering Report ====")
print(f"Total features generated: {report['total_features_generated']} ")
print(f"\nFeature categories:")
for category, count in report['feature_categories'].items():
    print(f"  {category}: {count}")
print(f"\nTop 10 features:")
for feature, score in report['top_10_features']:
    print(f"  {feature}: {score:.4f}")

# Test with machine learning model
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report

X = feature_matrix
y = df['label']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

# Train model
rf = RandomForestClassifier(n_estimators=100, random_state=42)
rf.fit(X_train, y_train)

# Evaluate
y_pred = rf.predict(X_test)
print("\n==== Model Performance with Engineered Features ====")
print(classification_report(y_test, y_pred))

```



```

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.ensemble import RandomForestClassifier,
GradientBoostingClassifier
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import shap
from lime.lime_tabular import LimeTabularExplainer
import warnings
warnings.filterwarnings('ignore')

class ModelInterpretabilitySuite:
    """
    Comprehensive model interpretability toolkit for security ML models
    """
    def __init__(self, model=None, feature_names=None):
        self.model = model
        self.feature_names = feature_names
        self.explainers = {}
        self.explanations = {}

    def setup_explainers(self, X_train, feature_names=None):
        """Initialize various explainers"""
        if feature_names:
            self.feature_names = feature_names

        # SHAP explainer
        print("Setting up SHAP explainer...")
        if hasattr(self.model, 'predict_proba'):
            self.explainers['shap'] = shap.TreeExplainer(self.model)
        else:
            self.explainers['shap'] = shap.KernelExplainer(
                self.model.predict,
                shap.sample(X_train, 100)
            )

        # LIME explainer
        print("Setting up LIME explainer...")
        self.explainers['lime'] = LimeTabularExplainer(
            X_train.values if hasattr(X_train, 'values') else X_train,
            feature_names=self.feature_names,
            class_names=['Normal', 'Anomaly'],
            mode='classification'
        )

        print("Explainers ready!")

```

```

def explain_instance(self, instance, method='all'):
    """Explain a single prediction"""
    explanations = {}

    # Get prediction
    if hasattr(self.model, 'predict_proba'):
        prediction = self.model.predict_proba(instance.reshape(1, -1))
[0]
        predicted_class = np.argmax(prediction)
    else:
        predicted_class = self.model.predict(instance.reshape(1, -1))
[0]
        prediction = [1-predicted_class, predicted_class]

    explanations['prediction'] = {
        'class': predicted_class,
        'probability': prediction
    }

    # SHAP explanation
    if method in ['all', 'shap'] and 'shap' in self.explainers:
        shap_values = self.explainers['shap'].shap_values(instance)
        if isinstance(shap_values, list):
            shap_values = shap_values[1] # For binary classification

        explanations['shap'] = {
            'values': shap_values,
            'base_value': self.explainers['shap'].expected_value[1]
            if
isinstance(self.explainers['shap'].expected_value, list)
                else self.explainers['shap'].expected_value,
            'feature_importance': dict(zip(self.feature_names,
shap_values))
        }
    }

    # LIME explanation
    if method in ['all', 'lime'] and 'lime' in self.explainers:
        lime_exp = self.explainers['lime'].explain_instance(
            instance,
            self.model.predict_proba if hasattr(self.model,
'predict_proba') else self.model.predict,
            num_features=10
        )

        explanations['lime'] = {
            'explanation': lime_exp.as_list(),
            'local_pred': lime_exp.local_pred
        }

```

```

        return explanations

    def explain_model_global(self, X, sample_size=1000):
        """Generate global model explanations"""
        print("\nGenerating global explanations...")

        # Sample data if too large
        if len(X) > sample_size:
            sample_idx = np.random.choice(len(X), sample_size,
replace=False)
            X_sample = X.iloc[sample_idx] if hasattr(X, 'iloc') else
X[sample_idx]
        else:
            X_sample = X

        # Global SHAP values
        print("Calculating global SHAP values...")
        shap_values = self.explainers['shap'].shap_values(X_sample)
        if isinstance(shap_values, list):
            shap_values = shap_values[1]

        # Feature importance
        feature_importance = np.abs(shap_values).mean(axis=0)
        importance_df = pd.DataFrame({
            'feature': self.feature_names,
            'importance': feature_importance
        }).sort_values('importance', ascending=False)

        self.explanations['global'] = {
            'feature_importance': importance_df,
            'shap_values': shap_values,
            'sample_data': X_sample
        }

        return importance_df

    def visualize_explanation(self, instance_idx, X,
explanation_type='shap'):
        """Visualize explanation for a specific instance"""
        instance = X.iloc[instance_idx] if hasattr(X, 'iloc') else
X[instance_idx]
        explanation = self.explain_instance(instance.values if
hasattr(instance, 'values') else instance)

        fig, axes = plt.subplots(2, 2, figsize=(15, 10))

        # Prediction details
        ax = axes[0, 0]
        ax.text(0.1, 0.8, f"Predicted Class: {explanation['prediction']}")
        [ 'class' ]}"]

```

```

        transform=ax.transAxes, fontsize=14)
    ax.text(0.1, 0.6, f"Probability: {explanation['prediction']"
['probability'][1]:.3f}",
            transform=ax.transAxes, fontsize=14)
    ax.axis('off')
    ax.set_title('Prediction Details')

    # SHAP waterfall plot
    if 'shap' in explanation:
        ax = axes[0, 1]
        shap_values = explanation['shap']['values']
        feature_values = instance.values if hasattr(instance,
'velues') else instance

        # Get top features
        top_features_idx = np.argsort(np.abs(shap_values))[-10:]
        top_features = [self.feature_names[i] for i in
top_features_idx]
        top_shap_values = [shap_values[i] for i in top_features_idx]
        top_feature_values = [feature_values[i] for i in
top_features_idx]

        # Create waterfall effect
        cumulative = explanation['shap']['base_value']
        positions = []
        for i, (feat, val, feat_val) in enumerate(zip(top_features,
top_shap_values, top_feature_values)):
            positions.append(cumulative)
            cumulative += val

        # Plot bars
        colors = ['red' if v < 0 else 'blue' for v in top_shap_values]
        bars = ax.barh(range(len(top_features)), top_shap_values,
                      left=positions, color=colors, alpha=0.8)

        # Add feature names and values
        labels = [f'{feat} = {val:.2f}' for feat, val in
zip(top_features, top_feature_values)]
        ax.set_yticks(range(len(top_features)))
        ax.set_yticklabels(labels)
        ax.set_xlabel('SHAP Value')
        ax.set_title('SHAP Waterfall Plot')
        ax.axvline(x=explanation['shap']['base_value'], color='gray',
linestyle='--', alpha=0.5)

        # LIME explanation
        if 'lime' in explanation:
            ax = axes[1, 0]
            lime_exp = explanation['lime']['explanation']
            features = [f[0] for f in lime_exp]

```

```

weights = [f[1] for f in lime_exp]

colors = ['red' if w < 0 else 'green' for w in weights]
y_pos = np.arange(len(features))

ax.barh(y_pos, weights, color=colors, alpha=0.8)
ax.set_yticks(y_pos)
ax.set_yticklabels(features)
ax.set_xlabel('LIME Weight')
ax.set_title('LIME Explanation')
ax.axvline(x=0, color='black', linestyle='--', alpha=0.3)

# Feature values radar chart
ax = axes[1, 1]

# Select top features for radar chart
if 'shap' in explanation:
    top_n = 8
    top_features_idx = np.argsort(np.abs(shap_values))[-top_n:]

    # Normalize feature values for radar chart
    feature_values_subset = [feature_values[i] for i in
top_features_idx]
    feature_names_subset = [self.feature_names[i] for i in
top_features_idx]

    # Create radar chart
    angles = np.linspace(0, 2 * np.pi, len(feature_names_subset),
endpoint=False)

    # Normalize values to [0, 1]
    normalized_values = []
    for i, val in enumerate(feature_values_subset):
        if hasattr(self, 'scaler') and self.scaler:
            normalized_values.append((val - self.scaler.mean_[i]) /
self.scaler.scale_[i])
        else:
            normalized_values.append(val)

    # Close the plot
    angles = np.concatenate((angles, [angles[0]]))
    normalized_values = normalized_values + [normalized_values[0]]

    ax.plot(angles, normalized_values, 'o-', linewidth=2)
    ax.fill(angles, normalized_values, alpha=0.25)
    ax.set_xticks(angles[:-1])
    ax.set_xticklabels(feature_names_subset, size=8)
    ax.set_title('Feature Values (Top Features)')
    ax.grid(True)

```

```

plt.tight_layout()
plt.show()

def generate_counterfactuals(self, instance, n_counterfactuals=5,
target_class=None):
    """Generate counterfactual explanations"""
    print("\nGenerating counterfactual explanations...")

    instance_array = instance.values if hasattr(instance, 'values')
else instance
    current_pred = self.model.predict(instance_array.reshape(1, -1))
[0]

    if target_class is None:
        target_class = 1 - current_pred

    counterfactuals = []

    # Simple approach: perturb features to find minimal changes
    for _ in range(n_counterfactuals * 10): # Generate more
candidates
        perturbed = instance_array.copy()

        # Randomly select features to change
        n_changes = np.random.randint(1, min(5, len(instance_array)))
        features_to_change = np.random.choice(len(instance_array),
n_changes, replace=False)

        for feat_idx in features_to_change:
            # Perturb feature
            if np.random.random() > 0.5:
                perturbed[feat_idx] *= np.random.uniform(0.8, 1.2)
            else:
                perturbed[feat_idx] += np.random.normal(0, 0.1)

        # Check if prediction changed
        new_pred = self.model.predict(perturbed.reshape(1, -1))[0]
        if new_pred == target_class:
            # Calculate distance
            distance = np.linalg.norm(perturbed - instance_array)
            counterfactuals.append({
                'counterfactual': perturbed,
                'changes': features_to_change,
                'distance': distance,
                'prediction': new_pred
            })

        # Sort by distance and return top n
        counterfactuals.sort(key=lambda x: x['distance'])
    return counterfactuals[:n_counterfactuals]

```

```

def security_specific_explanations(self, instance, context=None):
    """Generate security-specific explanations"""
    explanation = self.explain_instance(instance)

    security_explanation = {
        'risk_assessment': {},
        'recommended_actions': [],
        'similar_threats': [],
        'confidence_analysis': {}
    }

    # Risk assessment based on feature importance
    if 'shap' in explanation:
        shap_values = explanation['shap']['values']

        # Identify high-risk features
        high_risk_features = []
        for i, (feat, val) in enumerate(zip(self.feature_names,
shap_values)):
            if val > np.percentile(np.abs(shap_values), 90):
                high_risk_features.append({
                    'feature': feat,
                    'impact': val,
                    'value': instance[i]
                })

        security_explanation['risk_assessment']['high_risk_features'] = high_risk_features
    else:
        security_explanation['risk_assessment'] = {}

    # Generate recommendations
    for feat_info in high_risk_features:
        if 'port' in feat_info['feature'].lower():
            security_explanation['recommended_actions'].append(
                f"Monitor traffic on port-related feature: {feat_info['feature']}"
            )
        elif 'bytes' in feat_info['feature'].lower():
            security_explanation['recommended_actions'].append(
                f"Investigate data volume: {feat_info['feature']}"
            )
            feat_info['value'] = round(feat_info['value'], 2)
        elif 'duration' in feat_info['feature'].lower():
            security_explanation['recommended_actions'].append(
                f"Check connection duration: {feat_info['feature']}"
            )
            feat_info['value'] = round(feat_info['value'], 2)

    # Confidence analysis
    if hasattr(self.model, 'predict_proba'):

```

```

        proba = self.model.predict_proba(instance.reshape(1, -1))[0]
        confidence = max(proba)
        security_explanation['confidence_analysis'] = {
            'confidence': confidence,
            'uncertainty': 1 - confidence,
            'recommendation': 'High confidence' if confidence > 0.8
        } else 'Requires manual review'

    }

    return security_explanation

def generate_interpretability_report(self, X_test, y_test,
sample_size=100):
    """Generate comprehensive interpretability report"""
    print("\n==== Generating Interpretability Report ===")

    report = {
        'model_type': type(self.model).__name__,
        'global_importance': None,
        'explanation_consistency': {},
        'counterfactual_analysis': {},
        'security_insights': []
    }

    # Global feature importance
    global_importance = self.explain_model_global(X_test,
sample_size=sample_size)
    report['global_importance'] = global_importance.head(10).to_dict()

    # Check explanation consistency
    print("\nChecking explanation consistency...")
    sample_indices = np.random.choice(len(X_test), min(20,
len(X_test)), replace=False)

    shap_importances = []
    lime_importances = []

    for idx in sample_indices:
        instance = X_test.iloc[idx] if hasattr(X_test, 'iloc') else
X_test[idx]
        exp = self.explain_instance(instance.values if
hasattr(instance, 'values') else instance)

        if 'shap' in exp:
            shap_imp = np.abs(exp['shap']['values'])
            shap_importances.append(shap_imp)

        if 'lime' in exp:
            lime_imp = dict(exp['lime']['explanation'])
            lime_importances.append(lime_imp)

    return report

```

```
# Calculate consistency metrics
if shap_importances:
    shap_array = np.array(shap_importances)
    consistency_score = np.mean([
        np.corrcoef(shap_array[i], shap_array[j])[0, 1]
        for i in range(len(shap_array))
        for j in range(i+1, len(shap_array))
    ])
    report['explanation_consistency']['shap_consistency'] =
consistency_score

    # Generate security insights
    anomaly_indices = np.where(y_test == 1)[0][:5] # Get first 5
anomalies

    for idx in anomaly_indices:
        if idx < len(X_test):
            instance = X_test.iloc[idx] if hasattr(X_test, 'iloc')
        else X_test[idx]
            instance_array = instance.values if hasattr(instance,
'velues') else instance

            security_exp =
self.security_specific_explanations(instance_array)
            report['security_insights'].append({
                'instance_id': int(idx),
                'risk_features':
security_exp['risk_assessment'].get('high_risk_features', [])[:3],
                'recommendations': security_exp['recommended_actions']
[:3]
            })
    return report

# Example usage
print("== Model Interpretability Suite Demo ==\n")

# Generate sample data
from sklearn.datasets import make_classification

X, y = make_classification(
    n_samples=1000,
    n_features=20,
    n_informative=15,
    n_redundant=5,
    n_clusters_per_class=2,
    flip_y=0.1,
    random_state=42
```

```
)  
  
# Create feature names  
feature_names = [f'feature_{i}' for i in range(X.shape[1])]  
  
# Add some interpretable feature names for demo  
feature_names[0] = 'bytes_sent'  
feature_names[1] = 'connection_duration'  
feature_names[2] = 'packet_count'  
feature_names[3] = 'error_rate'  
feature_names[4] = 'port_scan_score'  
  
# Convert to DataFrame  
df = pd.DataFrame(X, columns=feature_names)  
df['target'] = y  
  
# Split data  
X_train, X_test, y_train, y_test = train_test_split(  
    df.drop('target', axis=1),  
    df['target'],  
    test_size=0.3,  
    random_state=42  
)  
  
# Train model  
print("Training Random Forest model...")  
model = RandomForestClassifier(n_estimators=100, random_state=42)  
model.fit(X_train, y_train)  
  
# Initialize interpretability suite  
interpreter = ModelInterpretabilitySuite(model, feature_names)  
interpreter.setup_explainers(X_train)  
  
# Global explanations  
global_importance = interpreter.explain_model_global(X_test,  
sample_size=300)  
print("\nTop 10 Most Important Features:")  
print(global_importance.head(10))  
  
# Explain specific instances  
print("\n==== Instance Explanations ===")  
  
# Find an anomaly  
anomaly_idx = np.where(y_test == 1)[0][0]  
print(f"\nExplaining anomaly at index {anomaly_idx}")  
interpreter.visualize_explanation(anomaly_idx, X_test)  
  
# Generate counterfactuals  
print("\n==== Counterfactual Analysis ===")  
instance = X_test.iloc[anomaly_idx]
```

```

counterfactuals = interpreter.generate_counterfactuals(instance,
n_counterfactuals=3)

print(f"Found {len(counterfactuals)} counterfactuals")
for i, cf in enumerate(counterfactuals):
    print(f"\nCounterfactual {i+1}:")
    print(f"  Distance from original: {cf['distance']:.3f}")
    print(f"  Features changed: {[feature_names[idx] for idx in
cf['changes']]})")

# Security-specific explanation
print("\n==== Security Analysis ===")
security_exp = interpreter.security_specific_explanations(instance.values)
print(f"High-risk features:
{len(security_exp['risk_assessment'].get('high_risk_features', []))}")
print(f"Recommended actions: {security_exp['recommended_actions'][:3]}")

# Generate comprehensive report
report = interpreter.generate_interpretability_report(X_test, y_test)
print("\n==== Interpretability Report Summary ===")
print(f"Model type: {report['model_type']}")
print(f"Explanation consistency (SHAP):
{report['explanation_consistency'].get('shap_consistency', 'N/A'):.3f}")
print(f"Security insights generated: {len(report['security_insights'])}")

# Visualize global feature importance
plt.figure(figsize=(10, 6))
importance_df = global_importance.head(15)
plt.barh(importance_df['feature'], importance_df['importance'])
plt.xlabel('Importance Score')
plt.title('Global Feature Importance (SHAP)')
plt.tight_layout()
plt.show()

```

# Section 4: Deep Learning Essentials - Solutions

## Exercise 1: Custom Loss Function

**Task:** Implement a custom loss function that combines MSE and MAE with a learnable weight.

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import numpy as np
import matplotlib.pyplot as plt

class AdaptiveLoss(keras.losses.Loss):
    """
    Custom loss that combines MSE and MAE with a learnable weight.
    Adapts based on error distribution.
    """
    def __init__(self, name='adaptive_loss'):
        super().__init__(name=name)
        # Learnable weight parameter
        self.alpha = tf.Variable(0.5, trainable=True,
                               constraint=lambda x: tf.clip_by_value(x,
0.0, 1.0))

    def call(self, y_true, y_pred):
        # Calculate MSE and MAE
        mse = tf.reduce_mean(tf.square(y_true - y_pred))
        mae = tf.reduce_mean(tf.abs(y_true - y_pred))

        # Combine with learnable weight
        loss = self.alpha * mse + (1 - self.alpha) * mae

        # Adapt weight based on error distribution
        # If errors are large (outliers), increase MAE weight
        error_std = tf.math.reduce_std(tf.abs(y_true - y_pred))
        self.alpha.assign(tf.clip_by_value(
            self.alpha - 0.01 * error_std, 0.0, 1.0
        ))

        return loss

    # Test the custom loss function
def test_adaptive_loss():
    # Generate data with outliers
    np.random.seed(42)
```

```

n_samples = 1000
X = np.random.randn(n_samples, 10)
y = 3 * X[:, 0] - 2 * X[:, 1] + np.random.randn(n_samples) * 0.1

# Add outliers
outlier_idx = np.random.choice(n_samples, 50, replace=False)
y[outlier_idx] += np.random.randn(50) * 10

# Split data
split = int(0.8 * n_samples)
X_train, X_test = X[:split], X[split:]
y_train, y_test = y[:split], y[split:]

# Build model with custom loss
model_custom = keras.Sequential([
    layers.Dense(64, activation='relu', input_shape=(10,)),
    layers.Dropout(0.3),
    layers.Dense(32, activation='relu'),
    layers.Dense(1)
])

# Compile with custom loss
custom_loss = AdaptiveLoss()
model_custom.compile(
    optimizer='adam',
    loss=custom_loss,
    metrics=['mae', 'mse']
)

# Train model
history_custom = model_custom.fit(
    X_train, y_train,
    validation_split=0.2,
    epochs=50,
    batch_size=32,
    verbose=0
)

# Compare with standard losses
model_mse = keras.models.clone_model(model_custom)
model_mse.compile(optimizer='adam', loss='mse', metrics=['mae'])

model_mae = keras.models.clone_model(model_custom)
model_mae.compile(optimizer='adam', loss='mae', metrics=['mse'])

history_mse = model_mse.fit(X_train, y_train, validation_split=0.2,
                             epochs=50, batch_size=32, verbose=0)
history_mae = model_mae.fit(X_train, y_train, validation_split=0.2,
                            epochs=50, batch_size=32, verbose=0)

```

```

# Visualize results
fig, axes = plt.subplots(2, 2, figsize=(12, 10))

# Loss comparison
ax = axes[0, 0]
ax.plot(history_custom.history['loss'], label='Adaptive Loss',
        linewidth=2)
ax.plot(history_mse.history['loss'], label='MSE', linewidth=2)
ax.plot(history_mae.history['loss'], label='MAE', linewidth=2)
ax.set_xlabel('Epoch')
ax.set_ylabel('Training Loss')
ax.set_title('Loss Comparison')
ax.legend()
ax.grid(True, alpha=0.3)

# Alpha evolution
ax = axes[0, 1]
alpha_history = []
for epoch in range(50):
    alpha_history.append(float(custom_loss.alpha))
ax.plot(alpha_history, 'g-', linewidth=2)
ax.set_xlabel('Epoch')
ax.set_ylabel('Alpha (MSE weight)')
ax.set_title('Adaptive Weight Evolution')
ax.grid(True, alpha=0.3)

# Predictions vs outliers
ax = axes[1, 0]
predictions_custom = model_custom.predict(X_test, verbose=0).flatten()
predictions_mse = model_mse.predict(X_test, verbose=0).flatten()

ax.scatter(y_test, predictions_custom, alpha=0.6, label='Adaptive')
ax.scatter(y_test, predictions_mse, alpha=0.6, label='MSE')
ax.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()],
        'r--', label='Perfect')
ax.set_xlabel('True Values')
ax.set_ylabel('Predictions')
ax.set_title('Prediction Accuracy')
ax.legend()
ax.grid(True, alpha=0.3)

# Error distribution
ax = axes[1, 1]
errors_custom = np.abs(y_test - predictions_custom)
errors_mse = np.abs(y_test - predictions_mse)

ax.hist(errors_custom, bins=30, alpha=0.6, label='Adaptive',
density=True)
ax.hist(errors_mse, bins=30, alpha=0.6, label='MSE', density=True)
ax.set_xlabel('Absolute Error')

```

```
ax.set_ylabel('Density')
ax.set_title('Error Distribution')
ax.legend()
ax.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

# Print final metrics
print(f"Final Alpha: {float(custom_loss.alpha):.3f}")
print(f"Test MAE - Adaptive: {np.mean(errors_custom):.3f}")
print(f"Test MAE - MSE Loss: {np.mean(errors_mse):.3f}")

# Run the test
test_adaptive_loss()
```

---

**Task:** Create a system that automatically tests different network architectures.

```
import itertools
from sklearn.model_selection import KFold
import time

class NeuralArchitectureSearch:
    """
    Automated system for finding optimal neural network architecture
    """

    def __init__(self, input_shape, output_shape, task='classification'):
        self.input_shape = input_shape
        self.output_shape = output_shape
        self.task = task
        self.search_results = []

        # Define search space
        self.search_space = {
            'n_layers': [2, 3, 4, 5],
            'neurons_per_layer': [32, 64, 128, 256],
            'activation': ['relu', 'elu', 'leaky_relu'],
            'dropout_rate': [0.0, 0.2, 0.3, 0.5],
            'batch_norm': [True, False],
            'learning_rate': [0.001, 0.01, 0.1]
        }

    def build_model(self, config):
        """Build model from configuration"""
        model = keras.Sequential()

        # Input layer
        model.add(layers.Input(shape=self.input_shape))

        # Hidden layers
        for i in range(config['n_layers']):
            # Gradually decrease neurons
            neurons = int(config['neurons_per_layer'] / (i + 1))

            # Dense layer
            if config['activation'] == 'leaky_relu':
                model.add(layers.Dense(neurons))
                model.add(layers.LeakyReLU())
            else:
                model.add(layers.Dense(neurons,
                                      activation=config['activation']))

        # Batch normalization
        if config['batch_norm']:
```

```
        model.add(layers.BatchNormalization())

        # Dropout
        if config['dropout_rate'] > 0:
            model.add(layers.Dropout(config['dropout_rate']))

    # Output layer
    if self.task == 'classification':
        model.add(layers.Dense(self.output_shape,
activation='softmax'))
        loss = 'sparse_categorical_crossentropy'
        metrics = ['accuracy']
    else:
        model.add(layers.Dense(self.output_shape))
        loss = 'mse'
        metrics = ['mae']

    # Compile
    model.compile(
optimizer=keras.optimizers.Adam(learning_rate=config['learning_rate']),
        loss=loss,
        metrics=metrics
    )

    return model

def evaluate_architecture(self, config, X, y, cv_folds=3):
    """Evaluate architecture using cross-validation"""
    kfold = KFold(n_splits=cv_folds, shuffle=True, random_state=42)
    scores = []
    train_times = []

    for train_idx, val_idx in kfold.split(X):
        X_train, X_val = X[train_idx], X[val_idx]
        y_train, y_val = y[train_idx], y[val_idx]

        # Build and train model
        model = self.build_model(config)

        # Early stopping
        early_stop = keras.callbacks.EarlyStopping(
            monitor='val_loss',
            patience=5,
            restore_best_weights=True
        )

        # Train
        start_time = time.time()
        history = model.fit(
```

```

        X_train, y_train,
        validation_data=(X_val, y_val),
        epochs=30,
        batch_size=32,
        callbacks=[early_stop],
        verbose=0
    )
    train_time = time.time() - start_time

    # Get best score
    if self.task == 'classification':
        best_score = max(history.history['val_accuracy'])
    else:
        best_score = -min(history.history['val_loss'])

    scores.append(best_score)
    train_times.append(train_time)

    # Clear session to free memory
    keras.backend.clear_session()

    return {
        'config': config,
        'mean_score': np.mean(scores),
        'std_score': np.std(scores),
        'mean_train_time': np.mean(train_times),
        'model_size': self._calculate_model_size(config)
    }

def _calculate_model_size(self, config):
    """Estimate model size in parameters"""
    size = 0
    prev_neurons = self.input_shape[0]

    for i in range(config['n_layers']):
        neurons = int(config['neurons_per_layer'] / (i + 1))
        size += prev_neurons * neurons + neurons # weights + bias
        if config['batch_norm']:
            size += 4 * neurons # gamma, beta, mean, variance
        prev_neurons = neurons

    size += prev_neurons * self.output_shape + self.output_shape
    return size

def search(self, X, y, n_random_samples=20, top_k=5):
    """Perform architecture search"""
    print(f"Starting Neural Architecture Search...")
    print(f"Search space size: {np.prod([len(v) for v in
self.search_space.values()])}")
    print(f"Random sampling: {n_random_samples} architectures\n")

```

```

# Random search
    for i in range(n_random_samples):
        # Sample random configuration
        config = {
            key: np.random.choice(values)
            for key, values in self.search_space.items()
        }

        print(f"Evaluating architecture {i+1}/{n_random_samples}...")
        result = self.evaluate_architecture(config, X, y)
        self.search_results.append(result)

        print(f"Score: {result['mean_score']:.4f} ±
{result['std_score']:.4f}")
        print(f"Train time: {result['mean_train_time']:.2f}s")
        print(f"Model size: {result['model_size']} parameters\n")

    # Sort by score
    self.search_results.sort(key=lambda x: x['mean_score'],
reverse=True)

    # Return top architectures
    return self.search_results[:top_k]

```

390

```

def visualize_results(self):
    """Visualize search results"""
    fig, axes = plt.subplots(2, 2, figsize=(12, 10))

    # Score vs complexity
    ax = axes[0, 0]
    scores = [r['mean_score'] for r in self.search_results]
    sizes = [r['model_size'] for r in self.search_results]
    colors = [r['config']['n_layers'] for r in self.search_results]

    scatter = ax.scatter(sizes, scores, c=colors, cmap='viridis',
                         s=100, alpha=0.6)
    ax.set_xlabel('Model Size (parameters)')
    ax.set_ylabel('Validation Score')
    ax.set_title('Score vs Model Complexity')
    plt.colorbar(scatter, ax=ax, label='Number of Layers')
    ax.grid(True, alpha=0.3)

    # Training time distribution
    ax = axes[0, 1]
    train_times = [r['mean_train_time'] for r in self.search_results]
    ax.hist(train_times, bins=20, alpha=0.7, color='blue')
    ax.set_xlabel('Training Time (seconds)')
    ax.set_ylabel('Count')
    ax.set_title('Training Time Distribution')

```

```

ax.grid(True, alpha=0.3)

# Hyperparameter importance
ax = axes[1, 0]
# Calculate correlation with score for each hyperparameter
importances = {}
for param in ['n_layers', 'neurons_per_layer', 'learning_rate']:
    values = [r['config'][param] for r in self.search_results]
    if isinstance(values[0], str):
        continue
    correlation = np.corrcoef(values, scores)[0, 1]
    importances[param] = abs(correlation)

ax.bar(importances.keys(), importances.values(), color='green',
       alpha=0.7)
ax.set_xlabel('Hyperparameter')
ax.set_ylabel('Absolute Correlation with Score')
ax.set_title('Hyperparameter Importance')
ax.tick_params(axis='x', rotation=45)
ax.grid(True, alpha=0.3)

# Pareto frontier
ax = axes[1, 1]
# Find Pareto optimal architectures
pareto_front = []
for i, result in enumerate(self.search_results):
    is_pareto = True
    for j, other in enumerate(self.search_results):
        if i != j:
            if (other['mean_score'] > result['mean_score'] and
                other['model_size'] < result['model_size']):
                is_pareto = False
                break
    if is_pareto:
        pareto_front.append(result)

# Plot all points
ax.scatter(sizes, scores, alpha=0.3, label='All architectures')

# Highlight Pareto front
pareto_sizes = [r['model_size'] for r in pareto_front]
pareto_scores = [r['mean_score'] for r in pareto_front]
ax.scatter(pareto_sizes, pareto_scores, color='red', s=100,
           label='Pareto optimal', zorder=10)

ax.set_xlabel('Model Size (parameters)')
ax.set_ylabel('Validation Score')
ax.set_title('Pareto Frontier')
ax.legend()
ax.grid(True, alpha=0.3)

```

```
plt.tight_layout()
plt.show()

# Test the architecture search
def test_architecture_search():
    # Generate synthetic data
    from sklearn.datasets import make_classification
    X, y = make_classification(
        n_samples=1000,
        n_features=20,
        n_informative=15,
        n_classes=5,
        random_state=42
    )

    # Create searcher
    searcher = NeuralArchitectureSearch(
        input_shape=(20,),
        output_shape=5,
        task='classification'
    )

    # Perform search
    top_architectures = searcher.search(X, y, n_random_samples=20,
   top_k=5)

    # Visualize results
    searcher.visualize_results()

    # Print best architecture
    print("\n==== Best Architecture ===")
    best = top_architectures[0]
    print(f"Configuration: {best['config']}")
    print(f"Score: {best['mean_score']:.4f} ± {best['std_score']:.4f}")
    print(f"Model size: {best['model_size']} parameters")
    print(f"Training time: {best['mean_train_time']:.2f} seconds")

test_architecture_search()
```

**Task:** Implement a neural network with multiple advanced regularization techniques.

```
class AdvancedRegularizedNetwork:  
    """  
        Neural network with custom regularization techniques  
    """  
  
    def __init__(self, input_shape, output_shape):  
        self.input_shape = input_shape  
        self.output_shape = output_shape  
  
    def create_custom_dropout(self, rate_schedule):  
        """Create dropout that varies by layer depth"""  
        class ScheduledDropout(layers.Layer):  
            def __init__(self, rate_fn, **kwargs):  
                super().__init__(**kwargs)  
                self.rate_fn = rate_fn  
                self.step = 0  
  
            def call(self, inputs, training=None):  
                if training:  
                    rate = self.rate_fn(self.step)  
                    self.step += 1  
                    return tf.nn.dropout(inputs, rate)  
                return inputs  
  
        return ScheduledDropout(rate_schedule)  
  
    def create_gradient_clipper(self, clip_norm=1.0):  
        """Create custom gradient clipping callback"""  
        class GradientClipper(keras.callbacks.Callback):  
            def __init__(self, clip_norm):  
                self.clip_norm = clip_norm  
                self.gradient_norms = []  
  
            def on_train_batch_end(self, batch, logs=None):  
                gradients = []  
                for layer in self.model.layers:  
                    if layer.weights:  
                        grads = tf.gradients(  
                            self.model.total_loss,  
                            layer.weights[0])  
                        if grads[0] is not None:  
                            gradients.append(tf.norm(grads[0]))  
  
                if gradients:  
                    total_norm = tf.norm(gradients)  
                    self.gradient_norms.append(float(total_norm))
```

```

    return GradientClipper(clip_norm)

def build_model(self):
    """Build model with advanced regularization"""
    inputs = layers.Input(shape=self.input_shape)

    # First block with L2 regularization
    x = layers.Dense(
        256,
        kernel_regularizer=keras.regularizers.l2(0.01),
        bias_regularizer=keras.regularizers.l2(0.001)
    )(inputs)
    x = layers.BatchNormalization()(x)
    x = layers.Activation('relu')(x)

    # Custom dropout - higher rate for deeper layers
    dropout_schedule = lambda step: 0.2 + 0.1 * (step // 1000)
    x = self.create_custom_dropout(dropout_schedule)(x)

    # Second block with L1 regularization (sparsity)
    x = layers.Dense(
        128,
        kernel_regularizer=keras.regularizers.l1(0.001),
        activity_regularizer=keras.regularizers.l1(0.0001)
    )(x)
    x = layers.BatchNormalization()(x)
    x = layers.LeakyReLU(alpha=0.1)(x)
    x = layers.Dropout(0.3)(x)

    # Third block with elastic net (L1 + L2)
    x = layers.Dense(
        64,
        kernel_regularizer=keras.regularizers.l1_l2(l1=0.001, l2=0.01)
    )(x)
    x = layers.BatchNormalization()(x)
    x = layers.ELU()(x)
    x = layers.Dropout(0.4)(x)

    # Spectral normalization for final layer
    final_layer = layers.Dense(32)
    x = final_layer(x)
    x = layers.BatchNormalization()(x)
    x = layers.Activation('relu')(x)

    # Output
    outputs = layers.Dense(self.output_shape, activation='softmax')(x)

    model = keras.Model(inputs=inputs, outputs=outputs)

```

```

        return model

    def create_learning_rate_schedule(self):
        """Create advanced learning rate schedule"""
        def schedule(epoch, lr):
            # Warm-up for first 5 epochs
            if epoch < 5:
                return lr * (epoch + 1) / 5
            # Cosine annealing
            elif epoch < 50:
                return 0.001 * (1 + np.cos(np.pi * (epoch - 5) / 45)) / 2
            # Fine-tuning phase
            else:
                return 0.0001 * np.exp(-0.1 * (epoch - 50))

        return keras.callbacks.LearningRateScheduler(schedule)

    def train_with_comparison(self, X_train, y_train, X_test, y_test):
        """Train and compare with/without regularization"""
        # Model with all regularization
        model_reg = self.build_model()
        model_reg.compile(
            optimizer=keras.optimizers.Adam(learning_rate=0.001),
            loss='sparse_categorical_crossentropy',
            metrics=['accuracy']
        )

        # Model without regularization
        model_no_reg = self.build_simple_model()
        model_no_reg.compile(
            optimizer=keras.optimizers.Adam(learning_rate=0.001),
            loss='sparse_categorical_crossentropy',
            metrics=['accuracy']
        )

        # Callbacks
        lr_schedule = self.create_learning_rate_schedule()
        gradient_clipper = self.create_gradient_clipper(clip_norm=1.0)

        # Train regularized model
        print("Training with advanced regularization...")
        history_reg = model_reg.fit(
            X_train, y_train,
            validation_split=0.2,
            epochs=100,
            batch_size=32,
            callbacks=[lr_schedule, gradient_clipper],
            verbose=0
        )

```

```

# Train unregularized model
print("Training without regularization...")
history_no_reg = model_no_reg.fit(
    X_train, y_train,
    validation_split=0.2,
    epochs=100,
    batch_size=32,
    verbose=0
)

# Visualize comparison
self.visualize_regularization_effects(
    history_reg, history_no_reg,
    model_reg, model_no_reg,
    X_test, y_test,
    gradient_clipper
)

return model_reg, model_no_reg

def build_simple_model(self):
    """Build model without regularization for comparison"""
    model = keras.Sequential([
        layers.Dense(256, activation='relu',
input_shape=self.input_shape),
        layers.Dense(128, activation='relu'),
        layers.Dense(64, activation='relu'),
        layers.Dense(32, activation='relu'),
        layers.Dense(self.output_shape, activation='softmax')
    ])
    return model

def visualize_regularization_effects(self, history_reg,
history_no_reg,
model_reg, model_no_reg,
X_test, y_test, gradient_clipper):
    """Visualize the effects of regularization"""
    fig, axes = plt.subplots(2, 3, figsize=(15, 10))

    # Training curves
    ax = axes[0, 0]
    ax.plot(history_reg.history['loss'], label='Train (Reg)',
linewidth=2)
    ax.plot(history_reg.history['val_loss'], label='Val (Reg)',
linewidth=2)
    ax.plot(history_no_reg.history['loss'], '--', label='Train (No
Reg)', linewidth=2)
    ax.plot(history_no_reg.history['val_loss'], '--', label='Val (No
Reg)', linewidth=2)
    ax.set_xlabel('Epoch')

```

```

        ax.set_ylabel('Loss')
        ax.set_title('Training Curves')
        ax.legend()
        ax.grid(True, alpha=0.3)

    # Accuracy comparison
    ax = axes[0, 1]
    ax.plot(history_reg.history['val_accuracy'], label='Regularized',
    linewidth=2)
    ax.plot(history_no_reg.history['val_accuracy'], label='No
    Regularization', linewidth=2)
    ax.set_xlabel('Epoch')
    ax.set_ylabel('Validation Accuracy')
    ax.set_title('Validation Accuracy')
    ax.legend()
    ax.grid(True, alpha=0.3)

    # Weight distributions
    ax = axes[0, 2]
    weights_reg = []
    weights_no_reg = []
    for layer in model_reg.layers:
        if hasattr(layer, 'kernel'):
            weights_reg.extend(layer.kernel.numpy().flatten())
    for layer in model_no_reg.layers:
        if hasattr(layer, 'kernel'):
            weights_no_reg.extend(layer.kernel.numpy().flatten())

        ax.hist(weights_reg, bins=50, alpha=0.6, label='Regularized',
density=True)
        ax.hist(weights_no_reg, bins=50, alpha=0.6, label='No Reg',
density=True)
        ax.set_xlabel('Weight Value')
        ax.set_ylabel('Density')
        ax.set_title('Weight Distribution')
        ax.legend()
        ax.grid(True, alpha=0.3)

    # Learning rate schedule
    ax = axes[1, 0]
    lrs = []
    for epoch in range(100):
        lr = 0.001
        lr = self.create_learning_rate_schedule().schedule(epoch, lr)
        lrs.append(lr)
    ax.plot(lrs, 'g-', linewidth=2)
    ax.set_xlabel('Epoch')
    ax.set_ylabel('Learning Rate')
    ax.set_title('Learning Rate Schedule')
    ax.set_yscale('log')

```

```

        ax.grid(True, alpha=0.3)

        # Gradient norms
        ax = axes[1, 1]
        if hasattr(gradient_clipper, 'gradient_norms') and
gradient_clipper.gradient_norms:
            ax.plot(gradient_clipper.gradient_norms, 'r-', alpha=0.7)
            ax.axhline(y=1.0, color='black', linestyle='--', label='Clip
threshold')
            ax.set_xlabel('Training Step')
            ax.set_ylabel('Gradient Norm')
            ax.set_title('Gradient Norms During Training')
            ax.legend()
        ax.grid(True, alpha=0.3)

        # Generalization gap
        ax = axes[1, 2]
        gap_reg = np.array(history_reg.history['accuracy']) -
np.array(history_reg.history['val_accuracy'])
        gap_no_reg = np.array(history_no_reg.history['accuracy']) -
np.array(history_no_reg.history['val_accuracy'])

        ax.plot(gap_reg, label='Regularized', linewidth=2)
        ax.plot(gap_no_reg, label='No Regularization', linewidth=2)
        ax.set_xlabel('Epoch')
        ax.set_ylabel('Train - Val Accuracy')
        ax.set_title('Generalization Gap')
        ax.legend()
        ax.grid(True, alpha=0.3)

    plt.tight_layout()
    plt.show()

    # Print final metrics
    test_loss_reg, test_acc_reg = model_reg.evaluate(X_test, y_test,
verbose=0)
    test_loss_no_reg, test_acc_no_reg = model_no_reg.evaluate(X_test,
y_test, verbose=0)

    print(f"\n==== Final Test Performance ===")
    print(f"Regularized Model: {test_acc_reg:.4f}")
    print(f"No Regularization: {test_acc_no_reg:.4f}")
    print(f"Improvement: {(test_acc_reg - test_acc_no_reg) * 100:.2f}%")


# Test advanced regularization
def test_advanced_regularization():
    # Generate synthetic data
    from sklearn.datasets import make_classification
    X, y = make_classification()

```

```
n_samples=5000,  
n_features=50,  
n_informative=30,  
n_redundant=10,  
n_classes=10,  
flip_y=0.1, # Add noise  
random_state=42  
)  
  
# Split data  
split = int(0.8 * len(X))  
X_train, X_test = X[:split], X[split:]  
y_train, y_test = y[:split], y[split:]  
  
# Create and train network  
network = AdvancedRegularizedNetwork(  
    input_shape=(50,),  
    output_shape=10  
)  
  
model_reg, model_no_reg = network.train_with_comparison(  
    X_train, y_train, X_test, y_test  
)  
  
test_advanced_regularization()
```

**Task:** Build a system that detects anomalous network protocols in real-time.

```
import time
from collections import deque
import threading

class RealTimeAnomalyDetector:
    """
    Real-time anomaly detection system for network protocols
    """

    def __init__(self, model_path=None, window_size=100,
                 threshold_percentile=95):
        self.window_size = window_size
        self.threshold_percentile = threshold_percentile

        # Sliding windows for different metrics
        self.prediction_window = deque(maxlen=window_size)
        self.confidence_window = deque(maxlen=window_size)
        self.latency_window = deque(maxlen=window_size)
        self.anomaly_scores = deque(maxlen=window_size)

        # Statistics
        self.normal_patterns = {}
        self.anomaly_threshold = None
        self.is_adapting = True

        # Model
        self.model = None
        self.scaler = None

        # Thread safety
        self.lock = threading.Lock()

    def build_model(self, input_dim):
        """Build anomaly detection model"""
        # Encoder
        encoder_input = layers.Input(shape=(input_dim,))
        x = layers.Dense(64, activation='relu')(encoder_input)
        x = layers.BatchNormalization()(x)
        x = layers.Dense(32, activation='relu')(x)
        x = layers.BatchNormalization()(x)
        encoded = layers.Dense(16, activation='relu', name='encoded')(x)

        # Decoder
        x = layers.Dense(32, activation='relu')(encoded)
        x = layers.BatchNormalization()(x)
        x = layers.Dense(64, activation='relu')(x)
        x = layers.BatchNormalization()(x)
```

400

```
        decoded = layers.Dense(input_dim, activation='sigmoid')(x)

    # Autoencoder
    autoencoder = keras.Model(encoder_input, decoded)
    autoencoder.compile(optimizer='adam', loss='mse')

    # Separate encoder for feature extraction
    encoder = keras.Model(encoder_input, encoded)

    return autoencoder, encoder

def train_on_normal_data(self, X_normal):
    """Train model on normal traffic patterns"""
    from sklearn.preprocessing import StandardScaler

    # Scale data
    self.scaler = StandardScaler()
    X_scaled = self.scaler.fit_transform(X_normal)

    # Build and train model
    self.model, self.encoder = self.build_model(X_normal.shape[1])

    print("Training anomaly detector on normal patterns...")
    history = self.model.fit(
        X_scaled, X_scaled,
        epochs=50,
        batch_size=32,
        validation_split=0.2,
        verbose=0
    )

    # Calculate threshold
    predictions = self.model.predict(X_scaled, verbose=0)
    reconstruction_errors = np.mean((X_scaled - predictions) ** 2,
axis=1)
    self.anomaly_threshold = np.percentile(
        reconstruction_errors,
        self.threshold_percentile
    )

    # Store normal pattern statistics
    self.normal_patterns = {
        'mean': np.mean(X_scaled, axis=0),
        'std': np.std(X_scaled, axis=0),
        'encoded_mean': np.mean(self.encoder.predict(X_scaled,
verbose=0), axis=0)
    }

    print(f"Training complete. Anomaly threshold:
{self.anomaly_threshold:.4f}")
```

```
    return history

def detect_anomaly(self, sample):
    """Detect if sample is anomalous"""
    with self.lock:
        # Scale sample
        sample_scaled = self.scaler.transform(sample.reshape(1, -1))

        # Get reconstruction
        start_time = time.time()
        reconstruction = self.model.predict(sample_scaled, verbose=0)
        latency = (time.time() - start_time) * 1000 # ms

        # Calculate reconstruction error
        error = np.mean((sample_scaled - reconstruction) ** 2)

        # Determine if anomaly
        is_anomaly = error > self.anomaly_threshold

        # Calculate confidence
        if is_anomaly:
            confidence = min(error / self.anomaly_threshold, 2.0) /
2.0
        else:
            confidence = 1.0 - (error / self.anomaly_threshold)

        # Update windows
        self.prediction_window.append(is_anomaly)
        self.confidence_window.append(confidence)
        self.latency_window.append(latency)
        self.anomaly_scores.append(error)

    return {
        'is_anomaly': is_anomaly,
        'confidence': confidence,
        'reconstruction_error': error,
        'latency_ms': latency
    }

def adapt_to_new_patterns(self, adaptation_rate=0.01):
    """Adapt model to new normal patterns"""
    if not self.is_adapting or len(self.anomaly_scores) <
self.window_size:
        return

    with self.lock:
        # Get recent non-anomalous samples
        recent_errors = np.array(self.anomaly_scores)
        recent_normal_mask = recent_errors < self.anomaly_threshold
```

```

        if np.sum(recent_normal_mask) > self.window_size * 0.8:
            # Mostly normal traffic - adapt threshold
            new_threshold = np.percentile(
                recent_errors[recent_normal_mask],
                self.threshold_percentile
            )

            # Exponential moving average
            self.anomaly_threshold = (
                (1 - adaptation_rate) * self.anomaly_threshold +
                adaptation_rate * new_threshold
            )
    }

def get_system_status(self):
    """Get current system status"""
    if len(self.prediction_window) == 0:
        return None

    with self.lock:
        recent_anomalies = sum(self.prediction_window)
        anomaly_rate = recent_anomalies / len(self.prediction_window)

        status = {
            'anomaly_rate': anomaly_rate,
            'avg_confidence': np.mean(self.confidence_window),
            'avg_latency_ms': np.mean(self.latency_window),
            'current_threshold': self.anomaly_threshold,
            'samples_processed': len(self.prediction_window),
            'alert_level': self._calculate_alert_level(anomaly_rate)
        }

    return status
403

def _calculate_alert_level(self, anomaly_rate):
    """Calculate alert level based on anomaly rate"""
    if anomaly_rate < 0.05:
        return 'GREEN'
    elif anomaly_rate < 0.15:
        return 'YELLOW'
    elif anomaly_rate < 0.30:
        return 'ORANGE'
    else:
        return 'RED'

def visualize_real_time_monitoring(self, duration_seconds=30):
    """Visualize real-time monitoring dashboard"""
    from matplotlib.animation import FuncAnimation

    fig, axes = plt.subplots(2, 3, figsize=(15, 10))

```

```
# Initialize data
times = deque(maxlen=100)
anomaly_rates = deque(maxlen=100)
latencies = deque(maxlen=100)
thresholds = deque(maxlen=100)

start_time = time.time()

def update_dashboard(frame):
    # Generate synthetic traffic
    if np.random.random() < 0.1: # 10% anomaly rate
        # Anomalous pattern
        sample = np.random.randn(20) * 2 + 1
    else:
        # Normal pattern
        sample = np.random.randn(20) * 0.5

    # Detect anomaly
    result = self.detect_anomaly(sample)

    # Adapt to new patterns
    self.adapt_to_new_patterns()

    # Get system status
    status = self.get_system_status()

    if status is None:
        return

    # Update data
    current_time = time.time() - start_time
    times.append(current_time)
    anomaly_rates.append(status['anomaly_rate'])
    latencies.append(result['latency_ms'])
    thresholds.append(self.anomaly_threshold)

    # Clear axes
    for ax in axes.flat:
        ax.clear()

    # 1. Anomaly rate over time
    ax = axes[0, 0]
    ax.plot(times, anomaly_rates, 'b-', linewidth=2)
    ax.axhline(y=0.1, color='orange', linestyle='--',
label='Warning')
    ax.axhline(y=0.3, color='red', linestyle='--',
label='Critical')
    ax.set_xlabel('Time (s)')
    ax.set_ylabel('Anomaly Rate')
```

404

```

        ax.set_title(f'Anomaly Rate (Alert: {status["alert_level"]})')
        ax.set_ylim(0, 0.5)
        ax.legend()
        ax.grid(True, alpha=0.3)

    # 2. Latency monitoring
    ax = axes[0, 1]
    ax.plot(times, latencies, 'g-', linewidth=2)
    ax.set_xlabel('Time (s)')
    ax.set_ylabel('Latency (ms)')
    ax.set_title(f'Detection Latency (Avg: {status["avg_latency_ms"]:.2f}ms)')
    ax.grid(True, alpha=0.3)

    # 3. Reconstruction errors
    ax = axes[0, 2]
    recent_errors = list(self.anomaly_scores)[-50:]
    ax.hist(recent_errors, bins=20, alpha=0.7, color='blue')
    ax.axvline(x=self.anomaly_threshold, color='red',
               linestyle='--', label='Threshold')
    ax.set_xlabel('Reconstruction Error')
    ax.set_ylabel('Count')
    ax.set_title('Error Distribution')
    ax.legend()
    ax.grid(True, alpha=0.3)

    # 4. Threshold adaptation
    ax = axes[1, 0]
    ax.plot(times, thresholds, 'r-', linewidth=2)
    ax.set_xlabel('Time (s)')
    ax.set_ylabel('Anomaly Threshold')
    ax.set_title('Adaptive Threshold')
    ax.grid(True, alpha=0.3)

    # 5. Confidence distribution
    ax = axes[1, 1]
    recent_confidences = list(self.confidence_window)
    if recent_confidences:
        ax.hist(recent_confidences, bins=20, alpha=0.7,
color='green')
        ax.set_xlabel('Confidence')
        ax.set_ylabel('Count')
        ax.set_title(f'Detection Confidence (Avg: {status["avg_confidence"]:.2f})')
        ax.set_xlim(0, 1)
        ax.grid(True, alpha=0.3)

    # 6. System metrics
    ax = axes[1, 2]
    metrics = [

```

```
f"Samples Processed: {status['samples_processed']}",
f"Anomaly Rate: {status['anomaly_rate']:.2%}",
f"Alert Level: {status['alert_level']}",
f"Avg Latency: {status['avg_latency_ms']:.2f}ms",
f"Threshold: {status['current_threshold']:.4f}"
]

for i, metric in enumerate(metrics):
    ax.text(0.1, 0.9 - i*0.15, metric, transform=ax.transAxes,
            fontsize=12, verticalalignment='top')

# Color code by alert level
colors = {
    'GREEN': 'lightgreen',
    'YELLOW': 'yellow',
    'ORANGE': 'orange',
    'RED': 'lightcoral'
}
ax.set_facecolor(colors.get(status['alert_level'], 'white'))
ax.set_xlim(0, 1)
ax.set_ylim(0, 1)
ax.axis('off')
ax.set_title('System Status')

plt.suptitle('Real-time Anomaly Detection Dashboard',
             fontsize=16) 406

# Animate
anim = FuncAnimation(fig, update_dashboard, interval=200,
                     frames=duration_seconds*5)

plt.tight_layout()
plt.show()

return anim

# Test real-time anomaly detection
def test_real_time_detection():
    # Generate training data (normal patterns)
    np.random.seed(42)
    X_normal = np.random.randn(1000, 20) * 0.5

    # Create detector
    detector = RealTimeAnomalyDetector(
        window_size=100,
        threshold_percentile=95
    )

    # Train on normal data
    detector.train_on_normal_data(X_normal)
```

```
# Simulate real-time detection
print("\nSimulating real-time anomaly detection...")

# Test with mixed traffic
n_samples = 500
anomalies_detected = 0

for i in range(n_samples):
    # Generate sample
    if i % 50 < 5: # Burst of anomalies
        sample = np.random.randn(20) * 2 + 1
        expected = True
    else:
        sample = np.random.randn(20) * 0.5
        expected = False

    # Detect
    result = detector.detect_anomaly(sample)

    if result['is_anomaly']:
        anomalies_detected += 1

    # Periodic status update
    if (i + 1) % 100 == 0:
        status = detector.get_system_status()
        print(f"\nAfter {i+1} samples:")
        print(f"  Anomaly rate: {status['anomaly_rate']:.2%}")
        print(f"  Alert level: {status['alert_level']}")
        print(f"  Avg latency: {status['avg_latency_ms']:.2f}ms")

    # Visualize monitoring dashboard
print("\nLaunching real-time monitoring dashboard...")
detector.visualize_real_time_monitoring(duration_seconds=10)

test_real_time_detection()
```

**Task:** Create an ensemble model with different architectures for protocol classification.

```
class EnsembleProtocolClassifier:  
    """  
        Ensemble of different neural network architectures for robust  
        classification  
    """  
  
    def __init__(self, input_shape, n_classes):  
        self.input_shape = input_shape  
        self.n_classes = n_classes  
        self.models = {}  
        self.model_weights = {}  
        self.performance_history = {}  
  
    def create_models(self):  
        """Create diverse model architectures"""\n        # Model 1: Deep Dense Network  
        model1 = keras.Sequential([  
            layers.Input(shape=self.input_shape),  
            layers.Dense(256, activation='relu'),  
            layers.BatchNormalization(),  
            layers.Dropout(0.3),  
            layers.Dense(128, activation='relu'),  
            layers.BatchNormalization(),  
            layers.Dropout(0.3),  
            layers.Dense(64, activation='relu'),  
            layers.Dense(self.n_classes, activation='softmax')  
        ], name='deep_dense')  
  
        # Model 2: Wide Network  
        model2 = keras.Sequential([  
            layers.Input(shape=self.input_shape),  
            layers.Dense(512, activation='relu'),  
            layers.Dropout(0.5),  
            layers.Dense(512, activation='relu'),  
            layers.Dropout(0.5),  
            layers.Dense(self.n_classes, activation='softmax')  
        ], name='wide_network')  
  
        # Model 3: Residual-like Network  
        inputs = layers.Input(shape=self.input_shape)  
        x = layers.Dense(128, activation='relu')(inputs)  
        x = layers.BatchNormalization()(x)  
  
        # Residual block  
        residual = x  
        x = layers.Dense(128, activation='relu')(x)  
        x = layers.BatchNormalization()(x)
```

```

x = layers.Dense(128, activation='relu')(x)
x = layers.Add()([x, residual])
x = layers.Activation('relu')(x)

x = layers.Dense(64, activation='relu')(x)
outputs = layers.Dense(self.n_classes, activation='softmax')(x)
model3 = keras.Model(inputs, outputs, name='residual_network')

# Model 4: Feature Engineering Network
inputs = layers.Input(shape=self.input_shape)

# Multiple feature extractors
feat1 = layers.Dense(64, activation='tanh')(inputs)
feat2 = layers.Dense(64, activation='relu')(inputs)
feat3 = layers.Dense(64, activation='elu')(inputs)

# Concatenate features
combined = layers.concatenate([feat1, feat2, feat3])
x = layers.Dense(128, activation='relu')(combined)
x = layers.Dropout(0.4)(x)
outputs = layers.Dense(self.n_classes, activation='softmax')(x)
model4 = keras.Model(inputs, outputs, name='feature_engineering')

# Model 5: Attention-based Network
inputs = layers.Input(shape=self.input_shape)
x = layers.Dense(128, activation='relu')(inputs)

# Simple attention mechanism
attention = layers.Dense(128, activation='sigmoid')(x)
x = layers.Multiply()([x, attention])

x = layers.Dense(64, activation='relu')(x)
outputs = layers.Dense(self.n_classes, activation='softmax')(x)
model5 = keras.Model(inputs, outputs, name='attention_network')

# Compile all models
self.models = {
    'deep_dense': model1,
    'wide_network': model2,
    'residual_network': model3,
    'feature_engineering': model4,
    'attention_network': model5
}

for name, model in self.models.items():
    model.compile(
        optimizer='adam',
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy']
    )

```

```

# Initialize weights
self.model_weights[name] = 1.0 / len(self.models)
self.performance_history[name] = []

return self.models

def train_ensemble(self, X_train, y_train, X_val, y_val, epochs=50):
    """Train all models in the ensemble"""
    histories = {}

    for name, model in self.models.items():
        print(f"\nTraining {name}...")

        # Custom callback to track performance
        class PerformanceTracker(keras.callbacks.Callback):
            def __init__(self, ensemble, model_name):
                self.ensemble = ensemble
                self.model_name = model_name

            def on_epoch_end(self, epoch, logs=None):
                val_acc = logs.get('val_accuracy', 0)

                self.ensemble.performance_history[self.model_name].append(val_acc)

            history = model.fit(
                X_train, y_train,
                validation_data=(X_val, y_val),
                epochs=epochs,
                batch_size=32,
                callbacks=[PerformanceTracker(self, name)],
                verbose=0
            )

            histories[name] = history

            # Update model weight based on validation performance
            final_val_acc = history.history['val_accuracy'][-1]
            self.model_weights[name] = final_val_acc

            # Normalize weights
            total_weight = sum(self.model_weights.values())
            for name in self.model_weights:
                self.model_weights[name] /= total_weight

        return histories

def predict_ensemble(self, X, strategy='weighted_voting'):
    """Make predictions using ensemble"""
    predictions = {}

```

```

# Get predictions from all models
for name, model in self.models.items():
    predictions[name] = model.predict(X, verbose=0)

if strategy == 'weighted_voting':
    # Weighted average of probabilities
    ensemble_pred = np.zeros((X.shape[0], self.n_classes))
    for name, pred in predictions.items():
        ensemble_pred += self.model_weights[name] * pred

    return np.argmax(ensemble_pred, axis=1), ensemble_pred

elif strategy == 'majority_voting':
    # Simple majority voting
    votes = []
    for name, pred in predictions.items():
        votes.append(np.argmax(pred, axis=1))
    votes = np.array(votes)

    # Get most common prediction
    ensemble_pred = []
    for i in range(X.shape[0]):
        counts = np.bincount(votes[:, i])
        ensemble_pred.append(np.argmax(counts))

    return np.array(ensemble_pred), predictions

elif strategy == 'confidence_weighted':
    # Weight by prediction confidence
    ensemble_pred = np.zeros((X.shape[0], self.n_classes))

    for name, pred in predictions.items():
        # Use max probability as confidence
        confidence = np.max(pred, axis=1, keepdims=True)
        ensemble_pred += confidence * pred

    # Normalize
    ensemble_pred /= ensemble_pred.sum(axis=1, keepdims=True)

    return np.argmax(ensemble_pred, axis=1), ensemble_pred

def track_individual_performance(self, X_test, y_test):
    """Track performance of individual models"""
    performances = {}

    for name, model in self.models.items():
        predictions = np.argmax(model.predict(X_test, verbose=0),
                               axis=1)
        accuracy = np.mean(predictions == y_test)
        performances[name] = accuracy

```

```
    return performances

def retrain_underperforming_models(self, X_train, y_train, X_val,
y_val, performance_threshold=0.8):
    """Automatically retrain models that underperform"""
    performances = self.track_individual_performance(X_val, y_val)

    for name, accuracy in performances.items():
        if accuracy < performance_threshold:
            print(f"\nRetraining {name} (accuracy: {accuracy:.3f})...")

            # Adjust learning rate
            self.models[name].compile(
                optimizer=keras.optimizers.Adam(learning_rate=0.0001),
                loss='sparse_categorical_crossentropy',
                metrics=['accuracy']
            )

            # Retrain with data augmentation
            self.models[name].fit(
                X_train, y_train,
                validation_data=(X_val, y_val),
                epochs=30,
                batch_size=32,
                verbose=0
            )

def visualize_ensemble_performance(self, X_test, y_test):
    """Visualize ensemble performance and characteristics"""
    fig, axes = plt.subplots(2, 3, figsize=(15, 10))

    # 1. Individual model accuracies
    ax = axes[0, 0]
    performances = self.track_individual_performance(X_test, y_test)
    models = list(performances.keys())
    accuracies = list(performances.values())

    bars = ax.bar(models, accuracies, color='skyblue')

    # Highlight best and worst
    best_idx = np.argmax(accuracies)
    worst_idx = np.argmin(accuracies)
    bars[best_idx].set_color('green')
    bars[worst_idx].set_color('red')

    ax.set_ylabel('Test Accuracy')
    ax.set_title('Individual Model Performance')
```

```

        ax.tick_params(axis='x', rotation=45)
        ax.grid(True, alpha=0.3, axis='y')

    # 2. Model weights
    ax = axes[0, 1]
    weights = [self.model_weights[name] for name in models]
    ax.pie(weights, labels=models, autopct='%.1f%%', startangle=90)
    ax.set_title('Model Weights in Ensemble')

    # 3. Training curves
    ax = axes[0, 2]
    for name, history in self.performance_history.items():
        if history:
            ax.plot(history, label=name, linewidth=2)
    ax.set_xlabel('Epoch')
    ax.set_ylabel('Validation Accuracy')
    ax.set_title('Training Progress')
    ax.legend()
    ax.grid(True, alpha=0.3)

    # 4. Ensemble vs individual
    ax = axes[1, 0]

    # Get ensemble predictions with different strategies
    strategies = ['weighted_voting', 'majority_voting',
'confidence_weighted']
    ensemble_accs = []

    for strategy in strategies:
        pred, _ = self.predict_ensemble(X_test, strategy=strategy)
        acc = np.mean(pred == y_test)
        ensemble_accs.append(acc)

    # Plot comparison
    all_accs = accuracies + ensemble_accs
    all_labels = models + [f'Ensemble ({s})' for s in strategies]
    colors = ['skyblue'] * len(models) + ['gold', 'orange', 'coral']

    bars = ax.bar(range(len(all_labels)), all_accs, color=colors)
    ax.set_xticks(range(len(all_labels)))
    ax.set_xticklabels(all_labels, rotation=45)
    ax.set_ylabel('Test Accuracy')
    ax.set_title('Ensemble vs Individual Models')
    ax.grid(True, alpha=0.3, axis='y')

    # 5. Confusion matrix for ensemble
    ax = axes[1, 1]
    ensemble_pred, _ = self.predict_ensemble(X_test,
strategy='weighted_voting')

```

```

        from sklearn.metrics import confusion_matrix
        cm = confusion_matrix(y_test, ensemble_pred)

        im = ax.imshow(cm, cmap='Blues')
        ax.set_xlabel('Predicted')
        ax.set_ylabel('Actual')
        ax.set_title('Ensemble Confusion Matrix')
        plt.colorbar(im, ax=ax)

        # Add numbers
        for i in range(cm.shape[0]):
            for j in range(cm.shape[1]):
                ax.text(j, i, str(cm[i, j]), ha='center', va='center')

        # 6. Prediction confidence distribution
        ax = axes[1, 2]
        _, ensemble_probs = self.predict_ensemble(X_test,
        strategy='weighted_voting')
        max_probs = np.max(ensemble_probs, axis=1)

        ax.hist(max_probs, bins=30, alpha=0.7, color='green')
        ax.axvline(x=np.mean(max_probs), color='red', linestyle='--',
                    label=f'Mean: {np.mean(max_probs):.3f}')
        ax.set_xlabel('Prediction Confidence')
        ax.set_ylabel('Count')
        ax.set_title('Ensemble Confidence Distribution')
        ax.legend()
        ax.grid(True, alpha=0.3)

        plt.tight_layout()
        plt.show()

# Test ensemble protocol classifier
def test_ensemble_classifier():
    # Generate synthetic protocol data
    from sklearn.datasets import make_classification
    X, y = make_classification(
        n_samples=5000,
        n_features=20,
        n_informative=15,
        n_redundant=5,
        n_classes=5, # 5 protocol types
        n_clusters_per_class=2,
        flip_y=0.05,
        random_state=42
    )

    # Split data
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=42

```

```
)  
X_train, X_val, y_train, y_val = train_test_split(  
    X_train, y_train, test_size=0.2, random_state=42  
)  
  
# Create ensemble  
ensemble = EnsembleProtocolClassifier(  
    input_shape=(20,),  
    n_classes=5  
)  
  
# Create models  
ensemble.create_models()  
  
# Train ensemble  
print("Training ensemble models...")  
histories = ensemble.train_ensemble(X_train, y_train, X_val, y_val,  
epochs=30)  
  
# Check for underperforming models  
ensemble.retrain_underperforming_models(X_train, y_train, X_val,  
y_val)  
  
# Visualize performance  
ensemble.visualize_ensemble_performance(X_test, y_test)  
  
# Final evaluation  
ensemble_pred, _ = ensemble.predict_ensemble(X_test,  
strategy='weighted_voting')  
ensemble_acc = np.mean(ensemble_pred == y_test)  
  
print(f"\n==== Final Results ===")  
print(f"Ensemble Accuracy: {ensemble_acc:.4f}")  
print(f"Model Weights: {ensemble.model_weights}")  
  
test_ensemble_classifier()
```

## Exercise 1: Multi-Scale CNN

**Task:** Build a CNN that uses multiple kernel sizes in parallel.

```
class MultiScaleCNN:  
    """  
    CNN with multiple kernel sizes for comprehensive pattern detection  
    """  
  
    def __init__(self, input_shape, n_classes):  
        self.input_shape = input_shape  
        self.n_classes = n_classes  
        self.kernel_patterns = {}  
  
    def build_multi_scale_model(self):  
        """Build CNN with multiple kernel sizes"""  
        inputs = layers.Input(shape=self.input_shape)  
  
        # Different kernel sizes for different pattern scales  
        kernel_sizes = [3, 5, 7]  
        kernel_outputs = []  
  
        for kernel_size in kernel_sizes:  
            # Conv branch for each kernel size  
            conv = layers.Conv1D(  
                filters=64,  
                kernel_size=kernel_size,  
                padding='same',  
                activation='relu',  
                name=f'conv_k{kernel_size}'  
) (inputs)  
  
            # Batch normalization  
            conv = layers.BatchNormalization() (conv)  
  
            # Second conv layer  
            conv = layers.Conv1D(  
                filters=32,  
                kernel_size=kernel_size,  
                padding='same',  
                activation='relu',  
                name=f'conv2_k{kernel_size}'  
) (conv)  
  
            # Global max pooling  
            pooled = layers.GlobalMaxPooling1D()(conv)  
  
            kernel_outputs.append(pooled)
```

```

# Concatenate all kernel outputs
merged = layers.concatenate(kernel_outputs)

# Attention mechanism to weight kernel outputs
attention = layers.Dense(len(kernel_sizes), activation='softmax')
(merged)
attention = layers.RepeatVector(merged.shape[1])(attention)
attention = layers.Permute((2, 1))(attention)

# Apply attention weights
weighted = layers.Multiply()([merged, attention[:, :, 0:1]])

# Classification layers
x = layers.Dense(128, activation='relu')(weighted)
x = layers.Dropout(0.5)(x)
x = layers.Dense(64, activation='relu')(x)
x = layers.Dropout(0.3)(x)

outputs = layers.Dense(self.n_classes, activation='softmax')(x)

model = keras.Model(inputs=inputs, outputs=outputs)

return model

def extract_kernel_patterns(self, model, X_sample):
    """Visualize what each kernel size detects"""
    # Create intermediate models for each kernel branch
    kernel_models = {}

    for kernel_size in [3, 5, 7]:
        layer_name = f'conv_k{kernel_size}'
        layer_output = model.get_layer(layer_name).output
        kernel_model = keras.Model(
            inputs=model.input,
            outputs=layer_output
        )
        kernel_models[kernel_size] = kernel_model

    # Get activations
    activations = {}
    for kernel_size, kernel_model in kernel_models.items():
        activation = kernel_model.predict(X_sample[np.newaxis, ...],
verbose=0)
        activations[kernel_size] = activation[0]

    return activations

def build_single_kernel_baseline(self, kernel_size=5):
    """Build single-kernel CNN for comparison"""
    model = keras.Sequential([

```

```

        layers.Conv1D(64, kernel_size, padding='same',
activation='relu',
                           input_shape=self.input_shape),
        layers.BatchNormalization(),
        layers.Conv1D(32, kernel_size, padding='same',
activation='relu'),
        layers.BatchNormalization(),
        layers.GlobalMaxPooling1D(),
        layers.Dense(128, activation='relu'),
        layers.Dropout(0.5),
        layers.Dense(64, activation='relu'),
        layers.Dropout(0.3),
        layers.Dense(self.n_classes, activation='softmax')
    )

    return model

def compare_performance(self, X_train, y_train, X_test, y_test):
    """Compare multi-scale with single-kernel CNN"""
    # Multi-scale model
    print("Training multi-scale CNN...")
    multi_model = self.build_multi_scale_model()
    multi_model.compile(
        optimizer='adam',
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy']
    )

    history_multi = multi_model.fit(
        X_train, y_train,
        validation_split=0.2,
        epochs=30,
        batch_size=32,
        verbose=0
    )

    # Single-kernel models
    single_histories = {}
    single_models = {}

    for kernel_size in [3, 5, 7]:
        print(f"Training single-kernel CNN (k={kernel_size})...")
        single_model = self.build_single_kernel_baseline(kernel_size)
        single_model.compile(
            optimizer='adam',
            loss='sparse_categorical_crossentropy',
            metrics=['accuracy']
        )

        history = single_model.fit(

```

```

        X_train, y_train,
        validation_split=0.2,
        epochs=30,
        batch_size=32,
        verbose=0
    )

    single_histories[kernel_size] = history
    single_models[kernel_size] = single_model

return multi_model, single_models, history_multi, single_histories

def visualize_results(self, multi_model, single_models,
                      history_multi, single_histories,
                      X_test, y_test):
    """Visualize comparison results"""
    fig, axes = plt.subplots(2, 3, figsize=(15, 10))

    # 1. Training curves comparison
    ax = axes[0, 0]
    ax.plot(history_multi.history['val_accuracy'],
            label='Multi-scale', linewidth=2, color='red')

    colors = ['blue', 'green', 'orange']
    for (kernel_size, history), color in zip(single_histories.items(),
  colors):
        ax.plot(history.history['val_accuracy'],
                label=f'Single k={kernel_size}', linewidth=2,
                color=color, linestyle='--')

    ax.set_xlabel('Epoch')
    ax.set_ylabel('Validation Accuracy')
    ax.set_title('Model Comparison')
    ax.legend()
    ax.grid(True, alpha=0.3)

    # 2. Test accuracy comparison
    ax = axes[0, 1]

    # Evaluate all models
    multi_acc = multi_model.evaluate(X_test, y_test, verbose=0)[1]
    single_accs = {}
    for kernel_size, model in single_models.items():
        single_accs[kernel_size] = model.evaluate(X_test, y_test,
  verbose=0)[1]

    models = ['Multi-scale'] + [f'k={k}' for k in single_accs.keys()]
    accuracies = [multi_acc] + list(single_accs.values())

    bars = ax.bar(models, accuracies, color=['red'] + colors)

```

```

        ax.set_ylabel('Test Accuracy')
        ax.set_title('Final Performance')
        ax.grid(True, alpha=0.3, axis='y')

        # Add values on bars
        for bar, acc in zip(bars, accuracies):
            ax.text(bar.get_x() + bar.get_width()/2, bar.get_height() +
0.01,
                    f'{acc:.3f}', ha='center', va='bottom')

    # 3. Kernel activation patterns
    ax = axes[0, 2]

    # Get sample and visualize activations
    sample_idx = 0
    X_sample = X_test[sample_idx]
    activations = self.extract_kernel_patterns(multi_model, X_sample)

    # Plot average activation strength
    kernel_sizes = list(activations.keys())
    avg_activations = [np.mean(np.abs(act)) for act in
activations.values()]

    ax.bar(kernel_sizes, avg_activations, color='purple', alpha=0.7)
    ax.set_xlabel('Kernel Size')
    ax.set_ylabel('Average Activation')
    ax.set_title(f'Kernel Responses (Class: {y_test[sample_idx]})')
    ax.grid(True, alpha=0.3, axis='y')

    # 4. Feature maps visualization
    for idx, (kernel_size, activation) in
enumerate(activations.items()):
        ax = axes[1, idx]

        # Show top 10 filters
        im = ax.imshow(activation[:, :10].T, aspect='auto',
cmap='hot')
        ax.set_xlabel('Sequence Position')
        ax.set_ylabel('Filter Index')
        ax.set_title(f'Kernel Size {kernel_size} Activations')
        plt.colorbar(im, ax=ax)

    plt.tight_layout()
    plt.show()

    # Get attention weights
    attention_layer = None
    for layer in multi_model.layers:
        if isinstance(layer, layers.Dense) and layer.output_shape[-1]
== 3:

```

```

        attention_layer = layer
        break

    if attention_layer:
        # Create model to extract attention weights
        attention_model = keras.Model(
            inputs=multi_model.input,
            outputs=attention_layer.output
        )

        # Visualize attention distribution
        fig, ax = plt.subplots(1, 1, figsize=(8, 6))

        # Get attention for multiple samples
        n_samples = min(100, len(X_test))
        attention_weights =
attention_model.predict(X_test[:n_samples], verbose=0)

        # Average attention per class
        for class_idx in range(self.n_classes):
            class_mask = y_test[:n_samples] == class_idx
            if np.any(class_mask):
                class_attention =
attention_weights[class_mask].mean(axis=0)
                ax.plot([3, 5, 7], class_attention, 'o-',
label=f'Class {class_idx}', markersize=8)

        ax.set_xlabel('Kernel Size')
        ax.set_ylabel('Average Attention Weight')
        ax.set_title('Kernel Importance by Class')
        ax.legend()
        ax.grid(True, alpha=0.3)
        plt.show()

# Test multi-scale CNN
def test_multi_scale_cnn():
    # Generate synthetic sequential data (e.g., byte sequences)
    np.random.seed(42)
    n_samples = 2000
    seq_length = 100
    n_classes = 4

    # Create patterns at different scales
    X = []
    y = []

    for i in range(n_samples):
        class_idx = i % n_classes

        # Base random sequence

```

```

    seq = np.random.randn(seq_length, 8)

    # Add class-specific patterns at different scales
    if class_idx == 0:
        # Small-scale pattern (size 3)
        pattern = np.array([1, -1, 1])
        for j in range(0, seq_length-3, 10):
            seq[j:j+3, 0] += pattern

    elif class_idx == 1:
        # Medium-scale pattern (size 5)
        pattern = np.sin(np.linspace(0, np.pi, 5))
        for j in range(0, seq_length-5, 15):
            seq[j:j+5, 1] += pattern

    elif class_idx == 2:
        # Large-scale pattern (size 7)
        pattern = np.array([0, 0.5, 1, 1.5, 1, 0.5, 0])
        for j in range(0, seq_length-7, 20):
            seq[j:j+7, 2] += pattern

    else:
        # Mixed patterns
        seq += 0.5 * np.sin(np.linspace(0, 4*np.pi, seq_length))[:,,
np.newaxis]

        X.append(seq)
        y.append(class_idx)

X = np.array(X)
y = np.array(y)

# Split data
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Create and test multi-scale CNN
multi_cnn = MultiScaleCNN(
    input_shape=(seq_length, 8),
    n_classes=n_classes
)

# Compare performance
multi_model, single_models, history_multi, single_histories =
    multi_cnn.compare_performance(X_train, y_train, X_test, y_test)

# Visualize results
multi_cnn.visualize_results(
    multi_model, single_models,

```

```
    history_multi, single_histories,  
    X_test, y_test  
)  
  
test_multi_scale_cnn()
```

---

**Task:** Create a denoising autoencoder that learns to reconstruct clean data from noisy input.

```
class DenoisingAutoencoderSystem:
    """
    Denoising autoencoder with multiple noise types
    """

    def __init__(self, input_dim):
        self.input_dim = input_dim
        self.noise_types = ['gaussian', 'dropout', 'adversarial',
'salt_pepper']
        self.models = {}

    def add_noise(self, data, noise_type, noise_level):
        """Add different types of noise to data"""
        noisy_data = data.copy()

        if noise_type == 'gaussian':
            # Additive Gaussian noise
            noise = np.random.normal(0, noise_level, data.shape)
            noisy_data += noise

        elif noise_type == 'dropout':
            # Random dropout noise
            mask = np.random.random(data.shape) > noise_level
            noisy_data *= mask

        elif noise_type == 'salt_pepper':
            # Salt and pepper noise
            salt_mask = np.random.random(data.shape) < noise_level/2
            pepper_mask = np.random.random(data.shape) < noise_level/2
            noisy_data[salt_mask] = 1.0
            noisy_data[pepper_mask] = -1.0

        elif noise_type == 'adversarial':
            # Simple adversarial perturbation
            gradient = np.sign(data - 0.5)
            noisy_data += noise_level * gradient

    return noisy_data

    def build_denoising_autoencoder(self, noise_type):
        """Build denoising autoencoder for specific noise type"""
        # Encoder
        encoder_input = layers.Input(shape=(self.input_dim,))

        # Deep encoder with skip connections
        e1 = layers.Dense(256, activation='relu')(encoder_input)
        e1 = layers.BatchNormalization()(e1)
```

424

```

e1 = layers.Dropout(0.2)(e1)

e2 = layers.Dense(128, activation='relu')(e1)
e2 = layers.BatchNormalization()(e2)

e3 = layers.Dense(64, activation='relu')(e2)
e3 = layers.BatchNormalization()(e3)

# Bottleneck
encoded = layers.Dense(32, activation='relu', name='bottleneck')
(e3)

# Decoder with skip connections
d1 = layers.Dense(64, activation='relu')(encoded)
d1 = layers.Add()([d1, e3]) # Skip connection
d1 = layers.BatchNormalization()(d1)

d2 = layers.Dense(128, activation='relu')(d1)
d2 = layers.Add()([d2, e2]) # Skip connection
d2 = layers.BatchNormalization()(d2)

d3 = layers.Dense(256, activation='relu')(d2)
d3 = layers.BatchNormalization()(d3)

# Output
decoded = layers.Dense(self.input_dim, activation='linear')(d3)

# Models
autoencoder = keras.Model(encoder_input, decoded)
encoder = keras.Model(encoder_input, encoded)

# Custom loss for denoising
if noise_type == 'adversarial':
    # Use robust loss for adversarial noise
    autoencoder.compile(
        optimizer='adam',
        loss='huber',
        metrics=['mae']
    )
else:
    autoencoder.compile(
        optimizer='adam',
        loss='mse',
        metrics=['mae']
    )

return autoencoder, encoder

def train_denoising_models(self, X_clean, noise_levels):
    """Train models for different noise types"""

```

```
histories = {}

    for noise_type in self.noise_types:
        print(f"\nTraining denoising autoencoder for {noise_type} noise...")

        # Build model
        autoencoder, encoder =
self.build_denoising_autoencoder(noise_type)

        # Generate noisy data
        X_noisy = self.add_noise(X_clean, noise_type,
noise_levels[noise_type])

        # Train
        history = autoencoder.fit(
            X_noisy, X_clean,
            validation_split=0.2,
            epochs=50,
            batch_size=32,
            verbose=0
        )

        self.models[noise_type] = {
            'autoencoder': autoencoder,
            'encoder': encoder,
            'history': history
        }

        histories[noise_type] = history

    return histories

def evaluate_robustness(self, X_test_clean, attack_types):
    """Evaluate robustness to different attack types"""
    results = {}

    for attack_type, attack_params in attack_types.items():
        print(f"\nEvaluating against {attack_type} attack...")

        # Generate attacked data
        if attack_type == 'fgsm':
            # Fast Gradient Sign Method
            epsilon = attack_params['epsilon']
            X_attacked = X_test_clean + epsilon * np.sign(
                np.random.randn(*X_test_clean.shape)
            )

        elif attack_type == 'pgd':
            # Projected Gradient Descent
```

```

        X_attacked = X_test_clean.copy()
        for _ in range(attack_params['iterations']):
            perturbation = attack_params['step_size'] * np.sign(
                np.random.randn(*X_test_clean.shape)
            )
            X_attacked += perturbation
            # Project back to valid range
            X_attacked = np.clip(X_attacked, -2, 2)

    elif attack_type == 'gaussian_blur':
        # Gaussian blur attack
        from scipy.ndimage import gaussian_filter1d
        X_attacked = gaussian_filter1d(
            X_test_clean,
            sigma=attack_params['sigma'],
            axis=1
        )

    else:
        X_attacked = X_test_clean

# Test each model
attack_results = {}
for noise_type, model_dict in self.models.items():
    autoencoder = model_dict['autoencoder']

    # Reconstruct
    X_reconstructed = autoencoder.predict(X_attacked,
verbose=0)

    # Calculate error
    reconstruction_error = np.mean((X_test_clean -
X_reconstructed) ** 2)
    detection_score = np.mean((X_attacked - X_reconstructed)
** 2, axis=1)

    attack_results[noise_type] = {
        'reconstruction_error': reconstruction_error,
        'detection_scores': detection_score,
        'detected_ratio': np.mean(detection_score >
np.percentile(detection_score, 95))
    }

    results[attack_type] = attack_results

return results

def visualize_denoising_results(self, X_clean, noise_levels):
    """Visualize denoising performance"""
    n_samples = 5

```

```

fig, axes = plt.subplots(len(self.noise_types), n_samples + 2,
                        figsize=(15, 12))

for i, noise_type in enumerate(self.noise_types):
    # Generate noisy samples
    X_noisy = self.add_noise(X_clean[:n_samples],
                             noise_type,
                             noise_levels=noise_type)

    # Get reconstructions
    autoencoder = self.models[noise_type]['autoencoder']
    X_reconstructed = autoencoder.predict(X_noisy, verbose=0)

    # Plot clean samples
    ax = axes[i, 0]
    ax.text(0.5, 0.5, f'{noise_type}\nNoise',
            ha='center', va='center', fontsize=12, weight='bold')
    ax.axis('off')

    # Plot samples
    for j in range(n_samples):
        ax = axes[i, j + 1]

        # Create visualization
        clean_sample = X_clean[j][:50] # First 50 features
        noisy_sample = X_noisy[j][:50]
        recon_sample = X_reconstructed[j][:50]

        x = np.arange(50)
        ax.plot(x, clean_sample, 'g-', label='Clean', alpha=0.8,
lineWidth=2)
        ax.plot(x, noisy_sample, 'r.', label='Noisy', alpha=0.5,
markerSize=4)
        ax.plot(x, recon_sample, 'b--', label='Reconstructed',
alpha=0.8, lineWidth=1.5)

        if i == 0 and j == 0:
            ax.legend(fontsize=8)

        ax.set_ylim(-3, 3)
        ax.set_xticks([])
        ax.set_yticks([])

    # Plot loss curve
    ax = axes[i, -1]
    history = self.models[noise_type]['history']
    ax.plot(history.history['loss'], label='Train')
    ax.plot(history.history['val_loss'], label='Val')
    ax.set_xlabel('Epoch')
    ax.set_ylabel('Loss')

```

```

        if i == 0:
            ax.legend()
            ax.grid(True, alpha=0.3)

plt.suptitle('Denoising Autoencoder Performance', fontsize=16)
plt.tight_layout()
plt.show()

# Robustness comparison
fig, axes = plt.subplots(1, 2, figsize=(12, 5))

# Noise level vs reconstruction error
ax = axes[0]
noise_ranges = np.linspace(0.1, 1.0, 10)

for noise_type in self.noise_types:
    errors = []
    autoencoder = self.models[noise_type]['autoencoder']

    for noise_level in noise_ranges:
        X_noisy = self.add_noise(X_clean[:100], noise_type,
noise_level)
        X_recon = autoencoder.predict(X_noisy, verbose=0)
        error = np.mean((X_clean[:100] - X_recon) ** 2)
        errors.append(error)

    ax.plot(noise_ranges, errors, 'o-', label=noise_type,
linewidth=2)

    ax.set_xlabel('Noise Level')
    ax.set_ylabel('Reconstruction Error')
    ax.set_title('Robustness to Increasing Noise')
    ax.legend()
    ax.grid(True, alpha=0.3)

# Cross-noise evaluation
ax = axes[1]
cross_results = np.zeros((len(self.noise_types),
len(self.noise_types)))

for i, train_noise in enumerate(self.noise_types):
    autoencoder = self.models[train_noise]['autoencoder']

    for j, test_noise in enumerate(self.noise_types):
        X_noisy = self.add_noise(X_clean[:100], test_noise, 0.3)
        X_recon = autoencoder.predict(X_noisy, verbose=0)
        error = np.mean((X_clean[:100] - X_recon) ** 2)
        cross_results[i, j] = error

im = ax.imshow(cross_results, cmap='RdYlGn_r')

```

```

        ax.set_xticks(range(len(self.noise_types)))
        ax.set_yticks(range(len(self.noise_types)))
        ax.set_xticklabels(self.noise_types, rotation=45)
        ax.set_yticklabels(self.noise_types)
        ax.set_xlabel('Test Noise Type')
        ax.set_ylabel('Trained on Noise Type')
        ax.set_title('Cross-Noise Generalization')
        plt.colorbar(im, ax=ax)

    # Add values
    for i in range(len(self.noise_types)):
        for j in range(len(self.noise_types)):
            ax.text(j, i, f'{cross_results[i, j]:.3f}', ha='center', va='center',
                    color='white' if cross_results[i, j] > 0.5 else
            'black')

    plt.tight_layout()
    plt.show()

# Test denoising autoencoder
def test_denoising_autoencoder():
    # Generate clean network traffic data
    np.random.seed(42)
    n_samples = 5000
    n_features = 100

    # Create structured data (network traffic patterns)
    X_clean = []

    for i in range(n_samples):
        # Base pattern
        if i % 3 == 0:
            # HTTP-like pattern
            pattern = np.sin(np.linspace(0, 4*np.pi, n_features)) * 0.5
            pattern[20:30] = 1.0 # Header spike
            pattern[60:65] = 0.8 # Data transfer
        elif i % 3 == 1:
            # SSH-like pattern
            pattern = np.random.randn(n_features) * 0.1
            pattern[::5] = 0.5 # Periodic keepalive
        else:
            # DNS-like pattern
            pattern = np.zeros(n_features)
            pattern[10] = 1.0 # Query
            pattern[15] = 0.8 # Response

        # Add some randomness
        pattern += np.random.randn(n_features) * 0.05
        X_clean.append(pattern)

```

```

X_clean = np.array(X_clean)

# Normalize
X_clean = (X_clean - X_clean.mean()) / X_clean.std()

# Split data
X_train = X_clean[:4000]
X_test = X_clean[4000:]

# Create denoising system
denoiser = DenoisingAutoencoderSystem(input_dim=n_features)

# Define noise levels
noise_levels = {
    'gaussian': 0.3,
    'dropout': 0.2,
    'salt_pepper': 0.1,
    'adversarial': 0.2
}

# Train models
histories = denoiser.train_denoising_models(X_train, noise_levels)

# Visualize results
denoiser.visualize_denoising_results(X_test, noise_levels) 431

# Test robustness
attack_types = {
    'fgsm': {'epsilon': 0.3},
    'pgd': {'iterations': 10, 'step_size': 0.1},
    'gaussian_blur': {'sigma': 2.0}
}

robustness_results = denoiser.evaluate_robustness(X_test,
attack_types)

# Print robustness summary
print("\n== Robustness Evaluation ==")
for attack_type, results in robustness_results.items():
    print(f"\n{attack_type} attack:")
    for model_type, metrics in results.items():
        print(f"  {model_type}: Error={metrics['reconstruction_error']:.4f}, "
              f"Detection={metrics['detected_ratio']:.2%}")

test_denoising_autoencoder()

```



**Task:** Design a hierarchical system with specialized autoencoders for each cluster.

```
class HierarchicalAutoencoderSystem:  
    """  
        Hierarchical autoencoder system with clustering and specialized models  
    """  
  
    def __init__(self, input_dim, n_clusters=3):  
        self.input_dim = input_dim  
        self.n_clusters = n_clusters  
        self.cluster_model = None  
        self.router_model = None  
        self.specialist_models = {}  
        self.cluster_thresholds = {}  
        self.cluster_stats = {}  
  
    def build_clustering_autoencoder(self):  
        """Build autoencoder for initial clustering"""  
        # Encoder  
        encoder_input = layers.Input(shape=(self.input_dim,))  
        x = layers.Dense(128, activation='relu')(encoder_input)  
        x = layers.BatchNormalization()(x)  
        x = layers.Dense(64, activation='relu')(x)  
        x = layers.BatchNormalization()(x)  
  
        # Latent representation for clustering  
        latent = layers.Dense(32, activation='relu', name='latent')(x)  
  
        # Clustering layer (learnable cluster centers)  
        clustering = layers.Dense(self.n_clusters, activation='softmax',  
                                  name='clustering')(latent)  
  
        # Decoder  
        x = layers.Dense(64, activation='relu')(latent)  
        x = layers.BatchNormalization()(x)  
        x = layers.Dense(128, activation='relu')(x)  
        x = layers.BatchNormalization()(x)  
        decoded = layers.Dense(self.input_dim, activation='linear')(x)  
  
        # Models  
        autoencoder = keras.Model(encoder_input, [decoded, clustering])  
        encoder = keras.Model(encoder_input, latent)  
        cluster_predictor = keras.Model(encoder_input, clustering)  
  
        return autoencoder, encoder, cluster_predictor  
  
    def build_specialist_autoencoder(self, cluster_id, cluster_size):  
        """Build specialized autoencoder for specific cluster"""  
        # Adjust architecture based on cluster characteristics
```

```

if cluster_size < 100:
    # Smaller architecture for small clusters
    hidden_sizes = [64, 32, 16]
elif cluster_size < 500:
    hidden_sizes = [128, 64, 32]
else:
    hidden_sizes = [256, 128, 64]

# Encoder
encoder_input = layers.Input(shape=(self.input_dim,))
x = encoder_input

for hidden_size in hidden_sizes:
    x = layers.Dense(hidden_size, activation='relu')(x)
    x = layers.BatchNormalization()(x)
    x = layers.Dropout(0.2)(x)

encoded = layers.Dense(hidden_sizes[-1]//2, activation='relu',
                      name=f'encoded_cluster_{cluster_id}')(x)

# Decoder
x = encoded
for hidden_size in reversed(hidden_sizes):
    x = layers.Dense(hidden_size, activation='relu')(x)
    x = layers.BatchNormalization()(x)

decoded = layers.Dense(self.input_dim, activation='linear')(x)

# Model
autoencoder = keras.Model(encoder_input, decoded)
autoencoder.compile(optimizer='adam', loss='mse', metrics=['mae'])

return autoencoder

def train_hierarchical_system(self, X_train, epochs_clustering=30,
                             epochs_specialist=20):
    """Train the complete hierarchical system"""
    print("Phase 1: Training clustering autoencoder...")

    # Build and train clustering model
    self.cluster_model, self.encoder, self.cluster_predictor = \
        self.build_clustering_autoencoder()

    # Custom loss for clustering
    def clustering_loss(y_true, y_pred):
        decoded, clustering = y_pred[0], y_pred[1]

        # Reconstruction loss
        recon_loss = tf.reduce_mean(tf.square(y_true - decoded))

        # Clustering loss
        clustering_loss = tf.reduce_mean(tf.square(clustering - y_true))

```

```
# Clustering loss (entropy regularization)
entropy = -tf.reduce_sum(clustering * tf.math.log(clustering +
1e-10),
                        axis=1)
entropy_loss = tf.reduce_mean(entropy)

return recon_loss + 0.1 * entropy_loss

self.cluster_model.compile(
    optimizer='adam',
    loss=clustering_loss
)

# Train clustering model
history_clustering = self.cluster_model.fit(
    X_train, X_train,
    epochs=epochs_clustering,
    batch_size=32,
    validation_split=0.2,
    verbose=0
)

# Assign clusters
cluster_probs = self.cluster_predictor.predict(X_train, verbose=0)
cluster_assignments = np.argmax(cluster_probs, axis=1) 435

print("\nPhase 2: Training specialist autoencoders...")

# Train specialist for each cluster
for cluster_id in range(self.n_clusters):
    cluster_mask = cluster_assignments == cluster_id
    cluster_data = X_train[cluster_mask]

    if len(cluster_data) < 10:
        print(f"Cluster {cluster_id} too small, skipping...")
        continue

    print(f"Training specialist for cluster {cluster_id} "
          f"({len(cluster_data)} samples)...")

# Build specialist
specialist = self.build_specialist_autoencoder(
    cluster_id, len(cluster_data)
)

# Train specialist
history = specialist.fit(
    cluster_data, cluster_data,
    epochs=epochs_specialist,
    batch_size=min(32, len(cluster_data)),
```

```

        validation_split=0.2,
        verbose=0
    )

    # Calculate threshold for anomaly detection
    predictions = specialist.predict(cluster_data, verbose=0)
    errors = np.mean((cluster_data - predictions) ** 2, axis=1)
    threshold = np.percentile(errors, 95)

    # Store specialist model and stats
    self.specialist_models[cluster_id] = specialist
    self.cluster_thresholds[cluster_id] = threshold
    self.cluster_stats[cluster_id] = {
        'size': len(cluster_data),
        'mean': np.mean(cluster_data, axis=0),
        'std': np.std(cluster_data, axis=0),
        'history': history
    }

print("\nPhase 3: Training router model...")

# Build router model (routes samples to appropriate specialist)
self.router_model = keras.Sequential([
    layers.Input(shape=(self.input_dim,)),
    layers.Dense(128, activation='relu'),
    layers.BatchNormalization(),
    layers.Dense(64, activation='relu'),
    layers.BatchNormalization(),
    layers.Dense(self.n_clusters, activation='softmax')
])

self.router_model.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)

# Train router
self.router_model.fit(
    X_train, cluster_assignments,
    epochs=20,
    batch_size=32,
    validation_split=0.2,
    verbose=0
)

return history_clustering

def detect_anomalies(self, X_test):
    """Detect anomalies using hierarchical system"""

```

```

results = {
    'cluster_anomalies': [],
    'specialist_anomalies': [],
    'combined_anomalies': [],
    'cluster_assignments': [],
    'reconstruction_errors': []
}

# Get cluster assignments
cluster_probs = self.router_model.predict(X_test, verbose=0)
cluster_assignments = np.argmax(cluster_probs, axis=1)
results['cluster_assignments'] = cluster_assignments

# Check each sample
for i, (sample, cluster_id) in enumerate(zip(X_test,
cluster_assignments)):
    sample_reshaped = sample.reshape(1, -1)

    # Cluster-level anomaly (low confidence in all clusters)
    max_cluster_prob = np.max(cluster_probs[i])
    is_cluster_anomaly = max_cluster_prob < 0.5
    results['cluster_anomalies'].append(is_cluster_anomaly)

    # Specialist-level anomaly
    if cluster_id in self.specialist_models:
        specialist = self.specialist_models[cluster_id]
        reconstruction = specialist.predict(sample_reshaped,
verbose=0)
        error = np.mean((sample_reshaped - reconstruction) ** 2)
        threshold = self.cluster_thresholds[cluster_id]

        is_specialist_anomaly = error > threshold

    results['specialist_anomalies'].append(is_specialist_anomaly)
    results['reconstruction_errors'].append(error)
else:
    # No specialist for this cluster
    results['specialist_anomalies'].append(True)
    results['reconstruction_errors'].append(np.inf)

    # Combined decision
    is_anomaly = is_cluster_anomaly or
results['specialist_anomalies'][-1]
    results['combined_anomalies'].append(is_anomaly)

return results

def adapt_clusters(self, X_new, adaptation_rate=0.1):
    """Adapt cluster assignments over time"""
    # Get current cluster assignments

```

```

cluster_probs = self.cluster_predictor.predict(X_new, verbose=0)
new_assignments = np.argmax(cluster_probs, axis=1)

# Check if any cluster is growing/shrinking significantly
for cluster_id in range(self.n_clusters):
    new_cluster_size = np.sum(new_assignments == cluster_id)
    old_cluster_size = self.cluster_stats.get(cluster_id,
{}).get('size', 0)

    if old_cluster_size > 0:
        size_change_ratio = new_cluster_size / old_cluster_size

        if size_change_ratio > 1.5 or size_change_ratio < 0.5:
            print(f"Cluster {cluster_id} size changed
significantly: "
f"{old_cluster_size} -> {new_cluster_size}")

    # Retrain specialist if needed
    if new_cluster_size > 10:
        cluster_data = X_new[new_assignments ==
cluster_id]
        self.specialist_models[cluster_id].fit(
            cluster_data, cluster_data,
            epochs=10,
            batch_size=min(32, len(cluster_data)),
            verbose=0
        )

        # Update threshold
        predictions =
self.specialist_models[cluster_id].predict(
            cluster_data, verbose=0
        )
        errors = np.mean((cluster_data - predictions) **

2, axis=1)

        # Exponential moving average for threshold
        new_threshold = np.percentile(errors, 95)
        old_threshold =
self.cluster_thresholds[cluster_id]
        self.cluster_thresholds[cluster_id] = (
            (1 - adaptation_rate) * old_threshold +
            adaptation_rate * new_threshold
        )

def visualize_hierarchical_system(self, X_test, y_test=None):
    """Visualize the hierarchical system performance"""
    # Get results
    results = self.detect_anomalies(X_test)

```

```
# Get latent representations
latent_representations = self.encoder.predict(X_test, verbose=0)

# Dimensionality reduction for visualization
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
latent_2d = pca.fit_transform(latent_representations)

fig, axes = plt.subplots(2, 3, figsize=(15, 10))

# 1. Cluster assignments
ax = axes[0, 0]
scatter = ax.scatter(latent_2d[:, 0], latent_2d[:, 1],
                     c=results['cluster_assignments'],
                     cmap='viridis', alpha=0.6)
ax.set_xlabel('Latent Dimension 1')
ax.set_ylabel('Latent Dimension 2')
ax.set_title('Cluster Assignments')
plt.colorbar(scatter, ax=ax)

# 2. Cluster-level anomalies
ax = axes[0, 1]
colors = ['blue' if not a else 'red'
          for a in results['cluster_anomalies']]
ax.scatter(latent_2d[:, 0], latent_2d[:, 1],
           c=colors, alpha=0.6)
ax.set_xlabel('Latent Dimension 1')
ax.set_ylabel('Latent Dimension 2')
ax.set_title('Cluster-level Anomalies (Red)')

# 3. Specialist-level anomalies
ax = axes[0, 2]
colors = ['blue' if not a else 'orange'
          for a in results['specialist_anomalies']]
ax.scatter(latent_2d[:, 0], latent_2d[:, 1],
           c=colors, alpha=0.6)
ax.set_xlabel('Latent Dimension 1')
ax.set_ylabel('Latent Dimension 2')
ax.set_title('Specialist-level Anomalies (Orange)')

# 4. Combined anomalies
ax = axes[1, 0]
colors = ['blue' if not a else 'red'
          for a in results['combined_anomalies']]
ax.scatter(latent_2d[:, 0], latent_2d[:, 1],
           c=colors, alpha=0.6)
ax.set_xlabel('Latent Dimension 1')
ax.set_ylabel('Latent Dimension 2')
ax.set_title('Combined Anomalies (Red)')
```

```

# 5. Reconstruction errors by cluster
ax = axes[1, 1]
for cluster_id in range(self.n_clusters):
    cluster_mask = np.array(results['cluster_assignments']) ==
cluster_id
        if np.any(cluster_mask):
            cluster_errors =
np.array(results['reconstruction_errors'])[cluster_mask]
            cluster_errors = cluster_errors[cluster_errors != np.inf]
            if len(cluster_errors) > 0:
                ax.hist(cluster_errors, bins=20, alpha=0.5,
label=f'Cluster {cluster_id}')

ax.set_xlabel('Reconstruction Error')
ax.set_ylabel('Count')
ax.set_title('Error Distribution by Cluster')
ax.legend()
ax.grid(True, alpha=0.3)

# 6. System statistics
ax = axes[1, 2]
stats_text = "Hierarchical System Statistics\n\n"

# Cluster sizes
for cluster_id in range(self.n_clusters):
    cluster_size = np.sum(
        np.array(results['cluster_assignments']) == cluster_id
    )
    stats_text += f"Cluster {cluster_id}: {cluster_size}
samples\n"

# Anomaly rates
cluster_anomaly_rate = np.mean(results['cluster_anomalies'])
specialist_anomaly_rate = np.mean(
    [a for a in results['specialist_anomalies'] if a != np.inf]
)
combined_anomaly_rate = np.mean(results['combined_anomalies'])

stats_text += f"\nAnomaly Rates:\n"
stats_text += f"Cluster-level: {cluster_anomaly_rate:.2%}\n"
stats_text += f"Specialist-level: {specialist_anomaly_rate:.2%}\n"
stats_text += f"Combined: {combined_anomaly_rate:.2%}\n"

ax.text(0.1, 0.9, stats_text, transform=ax.transAxes,
        fontsize=10, verticalalignment='top',
fontfamily='monospace')
ax.axis('off')

plt.suptitle('Hierarchical Autoencoder System Analysis',
fontsize=16)

```

```

plt.tight_layout()
plt.show()

# If true labels provided, show performance comparison
if y_test is not None:
    self.evaluate_performance(results, y_test)

def evaluate_performance(self, results, y_true):
    """Evaluate performance against true labels"""
    from sklearn.metrics import precision_recall_fscore_support,
roc_auc_score

        # Convert to binary arrays
        y_pred_cluster =
np.array(results['cluster_anomalies']).astype(int)
        y_pred_specialist =
np.array(results['specialist_anomalies']).astype(int)
        y_pred_combined =
np.array(results['combined_anomalies']).astype(int)

        # Calculate metrics
        methods = {
            'Cluster-level': y_pred_cluster,
            'Specialist-level': y_pred_specialist,
            'Combined': y_pred_combined
        }

        print("\n==== Performance Evaluation ===")
        for method_name, y_pred in methods.items():
            if np.all(y_pred == y_pred[0]): # Skip if all predictions are
same
                continue

            precision, recall, f1, _ = precision_recall_fscore_support(
                y_true, y_pred, average='binary'
            )

            try:
                auc = roc_auc_score(y_true, y_pred)
            except:
                auc = 0.0

            print(f"\n{method_name}:")
            print(f"  Precision: {precision:.3f}")
            print(f"  Recall: {recall:.3f}")
            print(f"  F1-Score: {f1:.3f}")
            print(f"  AUC: {auc:.3f}")

# Test hierarchical autoencoder
def test_hierarchical_autoencoder():

```

```

# Generate synthetic data with natural clusters
np.random.seed(42)
n_samples = 3000
n_features = 50

# Create 3 distinct traffic patterns
X = []
y = [] # True anomaly labels

# Cluster 0: Normal HTTP traffic
for i in range(1000):
    pattern = np.sin(np.linspace(0, 2*np.pi, n_features)) * 0.5
    pattern += np.random.randn(n_features) * 0.1
    X.append(pattern)
    y.append(0) # Normal

# Cluster 1: Normal SSH traffic
for i in range(1000):
    pattern = np.zeros(n_features)
    pattern[::5] = 1.0 # Periodic pattern
    pattern += np.random.randn(n_features) * 0.1
    X.append(pattern)
    y.append(0) # Normal

# Cluster 2: Normal DNS traffic
for i in range(800):
    pattern = np.zeros(n_features)
    pattern[np.random.choice(n_features, 5)] = 1.0 # Sparse spikes
    pattern += np.random.randn(n_features) * 0.1
    X.append(pattern)
    y.append(0) # Normal

# Add anomalies
# Type 1: Malformed HTTP
for i in range(100):
    pattern = np.sin(np.linspace(0, 4*np.pi, n_features)) * 2.0 # Wrong frequency
    pattern += np.random.randn(n_features) * 0.5
    X.append(pattern)
    y.append(1) # Anomaly

# Type 2: Suspicious SSH
for i in range(100):
    pattern = np.random.randn(n_features) * 1.5 # No structure
    X.append(pattern)
    y.append(1) # Anomaly

X = np.array(X)
y = np.array(y)

```

```
# Shuffle
indices = np.random.permutation(len(X))
X = X[indices]
y = y[indices]

# Split data
split = int(0.8 * len(X))
X_train = X[:split]
X_test = X[split:]
y_test = y[split:]

# Create and train system
hierarchical_system = HierarchicalAutoencoderSystem(
    input_dim=n_features,
    n_clusters=3
)

# Train
history = hierarchical_system.train_hierarchical_system(X_train)

# Test adaptation
print("\nTesting cluster adaptation...")
hierarchical_system.adapt_clusters(X_test[:100])

# Visualize results
hierarchical_system.visualize_hierarchical_system(X_test, y_test)

test_hierarchical_autoencoder()
```

**Task:** Implement a system that processes streaming log data with sliding windows.

```
import time
from collections import deque
import threading
import queue

class RealTimeLogMonitor:
    """
    Real-time log anomaly detection with LSTM autoencoder
    """

    def __init__(self, sequence_length=50, embedding_dim=32,
                 window_size=1000, update_interval=100):
        self.sequence_length = sequence_length
        self.embedding_dim = embedding_dim
        self.window_size = window_size
        self.update_interval = update_interval

        # Data structures
        self.log_buffer = deque(maxlen=window_size)
        self.anomaly_buffer = deque(maxlen=window_size)
        self.user_profiles = {}

        # Models
        self.tokenizer = None
        self.lstm_autoencoder = None
        self.threshold = None

        # Threading
        self.processing_queue = queue.Queue()
        self.is_running = False

    def build_lstm_autoencoder(self, vocab_size):
        """Build LSTM autoencoder for sequence modeling"""
        # Encoder
        encoder_inputs = layers.Input(shape=(self.sequence_length,))
        x = layers.Embedding(vocab_size, self.embedding_dim)
        (encoder_inputs)

        # Bidirectional LSTM encoder
        x, forward_h, forward_c, backward_h, backward_c =
        layers.Bidirectional(
            layers.LSTM(64, return_sequences=True, return_state=True))
        )(x)

        # Combine states
        state_h = layers.concatenate([forward_h, backward_h])
        state_c = layers.concatenate([forward_c, backward_c])
```

```

# Attention mechanism
attention = layers.MultiHeadAttention(
    num_heads=4, key_dim=64
)(x, x)

# Global context
context = layers.GlobalMaxPooling1D()(attention)

# Decoder
decoder_inputs = layers.RepeatVector(self.sequence_length)
(context)
decoder_lstm = layers.LSTM(128, return_sequences=True)(
    decoder_inputs, initial_state=[state_h, state_c]
)

# Output
decoder_outputs = layers.TimeDistributed(
    layers.Dense(vocab_size, activation='softmax')
)(decoder_lstm)

# Model
autoencoder = keras.Model(encoder_inputs, decoder_outputs)
autoencoder.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)

# Encoder model for feature extraction
encoder = keras.Model(encoder_inputs, context)

return autoencoder, encoder

def preprocess_logs(self, logs):
    """Preprocess log messages"""
    from sklearn.feature_extraction.text import CountVectorizer

    if self.tokenizer is None:
        self.tokenizer = CountVectorizer(
            max_features=10000,
            token_pattern=r'\b\w+\b',
            lowercase=True
        )
        self.tokenizer.fit(logs)

    # Simple tokenization (in practice, use more sophisticated
methods)
    processed = []
    for log in logs:

```

```

# Extract tokens
tokens = log.lower().split()

# Convert to indices
indices = []
for token in tokens[:self.sequence_length]:
    if token in self.tokenizer.vocabulary_:
        indices.append(self.tokenizer.vocabulary_[token])
    else:
        indices.append(0) # Unknown token

# Pad sequence
while len(indices) < self.sequence_length:
    indices.append(0)

processed.append(indices[:self.sequence_length])

return np.array(processed)

def train_on_normal_logs(self, normal_logs):
    """Train LSTM autoencoder on normal log patterns"""
    print("Training log anomaly detector...")

    # Preprocess logs
    X = self.preprocess_logs(normal_logs)

    # Build model
    vocab_size = len(self.tokenizer.vocabulary_) + 1
    self.lstm_autoencoder, self.encoder =
self.build_lstm_autoencoder(vocab_size)

    # Train
    history = self.lstm_autoencoder.fit(
        X, X,
        epochs=20,
        batch_size=32,
        validation_split=0.2,
        verbose=0
    )

    # Calculate threshold
    predictions = self.lstm_autoencoder.predict(X, verbose=0)

    # Calculate reconstruction error (cross-entropy)
    errors = []
    for i in range(len(X)):
        error = -np.sum(
            np.eye(vocab_size)[X[i]] * np.log(predictions[i] + 1e-7)
        ) / self.sequence_length
        errors.append(error)

```

```

        self.threshold = np.percentile(errors, 95)

    print(f"Training complete. Threshold: {self.threshold:.4f}")

    return history

def process_log_stream(self, log_entry):
    """Process incoming log entry"""
    # Extract user/system info
    user = self._extract_user(log_entry)

    # Add to buffer
    self.log_buffer.append({
        'timestamp': time.time(),
        'log': log_entry,
        'user': user
    })

    # Check if we should process
    if len(self.log_buffer) >= self.sequence_length:
        # Get recent logs
        recent_logs = [entry['log'] for entry in
                       list(self.log_buffer)[-self.sequence_length:]]
        # Preprocess
        X = self.preprocess_logs(recent_logs)

        # Predict
        prediction = self.lstm_autoencoder.predict(X[-1:], verbose=0)

        # Calculate error
        vocab_size = prediction.shape[-1]
        error = -np.sum(
            np.eye(vocab_size)[X[-1]] * np.log(prediction[0] + 1e-7)
        ) / self.sequence_length

        # Detect anomaly
        is_anomaly = error > self.threshold
        anomaly_score = error / self.threshold if self.threshold > 0
else 0

        # Update user profile
        self._update_user_profile(user, is_anomaly, anomaly_score)

        # Store result
        result = {
            'timestamp': time.time(),
            'log': log_entry,
            'user': user,

```

```

        'is_anomaly': is_anomaly,
        'anomaly_score': anomaly_score,
        'error': error
    }

    self.anomaly_buffer.append(result)

    # Trigger alert if needed
    if is_anomaly and anomaly_score > 2.0:
        self._trigger_alert(result)

    return result

return None

def _extract_user(self, log_entry):
    """Extract user from log entry"""
    # Simple extraction (customize based on log format)
    import re
    user_match = re.search(r'user[:\s]+(\w+)', log_entry.lower())
    return user_match.group(1) if user_match else 'unknown'

def _update_user_profile(self, user, is_anomaly, anomaly_score):
    """Update user behavior profile"""
    if user not in self.user_profiles:
        self.user_profiles[user] = {
            'total_logs': 0,
            'anomalies': 0,
            'avg_score': 0,
            'recent_scores': deque(maxlen=100)
        }

    profile = self.user_profiles[user]
    profile['total_logs'] += 1
    if is_anomaly:
        profile['anomalies'] += 1
    profile['recent_scores'].append(anomaly_score)
    profile['avg_score'] = np.mean(profile['recent_scores'])

def _trigger_alert(self, anomaly):
    """Trigger alert for significant anomaly"""
    print(f"\n⚠️ ALERT: High anomaly score detected!")
    print(f"User: {anomaly['user']}")
    print(f"Score: {anomaly['anomaly_score']:.2f}")
    print(f"Log: {anomaly['log'][:100]}...")

def get_interpretable_explanation(self, log_entry):
    """Provide interpretable explanation for anomaly"""
    # Preprocess log
    X = self.preprocess_logs([log_entry])

```

```

# Get reconstruction
prediction = self.lstm_autoencoder.predict(X, verbose=0)[0]

# Find most surprising tokens
vocab_size = prediction.shape[-1]
surprises = []

for i, token_id in enumerate(X[0]):
    if token_id > 0: # Skip padding
        expected_prob = prediction[i, token_id]
        surprise = -np.log(expected_prob + 1e-7)

        # Get token
        token = None
        for word, idx in self.tokenizer.vocabulary_.items():
            if idx == token_id:
                token = word
                break

        if token:
            surprises.append((token, surprise, i))

# Sort by surprise
surprises.sort(key=lambda x: x[1], reverse=True)

```

449

```

# Generate explanation
explanation = "Anomaly Explanation:\n"
explanation += f"Overall anomaly score:\n{self._calculate_anomaly_score(X[0], prediction):.2f}\n\n"
explanation += "Most surprising tokens:\n"

for token, surprise, position in surprises[:5]:
    explanation += f"  Position {position}: '{token}' (surprise: {surprise:.2f})\n"

return explanation

```

```

def _calculate_anomaly_score(self, sequence, prediction):
    """Calculate anomaly score for a sequence"""
    vocab_size = prediction.shape[-1]
    error = -np.sum(
        np.eye(vocab_size)[sequence] * np.log(prediction + 1e-7)
    ) / self.sequence_length
    return error / self.threshold if self.threshold > 0 else 0

```

```

def visualize_monitoring_dashboard(self, duration_seconds=30):
    """Real-time monitoring dashboard"""
    from matplotlib.animation import FuncAnimation

```

```
fig, axes = plt.subplots(2, 3, figsize=(15, 10))
```

```
# Initialize plot data
```

```
times = deque(maxlen=100)
```

```
anomaly_scores = deque(maxlen=100)
```

```
user_activity = {}
```

```
start_time = time.time()
```

```
def update_dashboard(frame):
```

```
    # Generate synthetic log
```

```
    log_entry = self._generate_synthetic_log()
```

```
    # Process log
```

```
    result = self.process_log_stream(log_entry)
```

```
    if result is None:
```

```
        return
```

```
    # Update data
```

```
    current_time = time.time() - start_time
```

```
    times.append(current_time)
```

```
    anomaly_scores.append(result['anomaly_score'])
```

```
    # Clear axes
```

```
    for ax in axes.flat:
```

```
        ax.clear()
```

```
    # 1. Anomaly score timeline
```

```
    ax = axes[0, 0]
```

```
    ax.plot(times, anomaly_scores, 'b-', linewidth=2)
```

```
    ax.axhline(y=1.0, color='orange', linestyle='--',
```

```
label='Threshold')
```

```
    ax.axhline(y=2.0, color='red', linestyle='--', label='Alert')
```

```
    ax.set_xlabel('Time (s)')
```

```
    ax.set_ylabel('Anomaly Score')
```

```
    ax.set_title('Real-time Anomaly Scores')
```

```
    ax.legend()
```

```
    ax.grid(True, alpha=0.3)
```

```
    # 2. User activity heatmap
```

```
    ax = axes[0, 1]
```

```
    # Aggregate user scores
```

```
    for entry in list(self.anomaly_buffer)[-50:]:
```

```
        user = entry['user']
```

```
        if user not in user_activity:
```

```
            user_activity[user] = []
```

```
        user_activity[user].append(entry['anomaly_score'])
```

450

```

if user_activity:
    users = list(user_activity.keys())[:10] # Top 10 users
    data = []
    for user in users:
        scores = user_activity[user][-20:] # Last 20
activities
        # Pad if needed
        while len(scores) < 20:
            scores = [0] + scores
        data.append(scores)

im = ax.imshow(data, aspect='auto', cmap='YlOrRd')
ax.set_yticks(range(len(users)))
ax.set_yticklabels(users)
ax.set_xlabel('Time Window')
ax.set_title('User Activity Heatmap')
plt.colorbar(im, ax=ax)

# 3. Recent anomalies
ax = axes[0, 2]
recent_anomalies = [e for e in list(self.anomaly_buffer)[-20:]
                     if e['is_anomaly']]

if recent_anomalies:
    y_pos = 0.9
    for anomaly in recent_anomalies[-5:]: # Show last 5
        ax.text(0.05, y_pos,
                f'{anomaly["user"]}: {anomaly["log"][:50]}...', 
                transform=ax.transAxes, fontsize=9,
                bbox=dict(boxstyle="round,pad=0.3",
                          facecolor='lightcoral', alpha=0.5))
    y_pos -= 0.18

    ax.set_xlim(0, 1)
    ax.set_ylimits(0, 1)
    ax.axis('off')
    ax.set_title('Recent Anomalies')

# 4. User risk scores
ax = axes[1, 0]
if self.user_profiles:
    users = list(self.user_profiles.keys())
    risk_scores = [profile['avg_score'] for profile in
                  self.user_profiles.values()]

bars = ax.bar(users[:10], risk_scores[:10])

# Color code by risk level
for bar, score in zip(bars, risk_scores[:10]):
    if score > 1.5:

```

```

                bar.set_color('red')
            elif score > 1.0:
                bar.set_color('orange')
            else:
                bar.set_color('green')

        ax.set_xlabel('User')
        ax.set_ylabel('Average Risk Score')
        ax.set_title('User Risk Profiles')
        ax.tick_params(axis='x', rotation=45)
        ax.grid(True, alpha=0.3, axis='y')

# 5. Anomaly distribution
ax = axes[1, 1]
if self.anomaly_buffer:
    scores = [e['anomaly_score'] for e in self.anomaly_buffer]
    ax.hist(scores, bins=30, alpha=0.7, color='blue')
    ax.axvline(x=1.0, color='orange', linestyle='--',
               label='Threshold')
    ax.axvline(x=2.0, color='red', linestyle='--',
               label='Alert Level')
    ax.set_xlabel('Anomaly Score')
    ax.set_ylabel('Count')
    ax.set_title('Score Distribution')
    ax.legend()
    ax.grid(True, alpha=0.3)

# 6. System metrics
ax = axes[1, 2]

total_logs = len(self.log_buffer)
total_anomalies = sum(1 for e in self.anomaly_buffer
                      if e['is_anomaly'])
anomaly_rate = total_anomalies / max(len(self.anomaly_buffer),
1)

metrics_text = f"System Metrics\n\n"
metrics_text += f"Logs Processed: {total_logs}\n"
metrics_text += f"Anomalies Detected: {total_anomalies}\n"
metrics_text += f"Anomaly Rate: {anomaly_rate:.2%}\n"
metrics_text += f"Active Users: {len(self.user_profiles)}\n"
metrics_text += f"Buffer Size:\n{len(self.log_buffer)}/{self.window_size}\n"

ax.text(0.1, 0.9, metrics_text, transform=ax.transAxes,
        fontsize=12, verticalalignment='top',
        fontfamily='monospace')
ax.axis('off')

plt.suptitle('Real-time Log Anomaly Detection Dashboard',

```

```

fontsize=16)

    # Animate
    anim = FuncAnimation(fig, update_dashboard, interval=500,
                         frames=duration_seconds*2)

    plt.tight_layout()
    plt.show()

    return anim

def _generate_synthetic_log(self):
    """Generate synthetic log entries for testing"""
    users = ['alice', 'bob', 'charlie', 'david', 'eve']
    actions = ['login', 'logout', 'access', 'modify', 'delete',
    'create']
    resources = ['file', 'database', 'server', 'config', 'system']

    # Normal log pattern
    if np.random.random() < 0.9:
        user = np.random.choice(users)
        action = np.random.choice(actions[:4]) # Normal actions
        resource = np.random.choice(resources)
        log = f"User: {user} Action: {action} Resource: {resource}"
    Status: success"
    else:
        # Anomalous patterns
        anomaly_type = np.random.choice(['unusual_user',
    'unusual_action',
                           'unusual_pattern'])

        if anomaly_type == 'unusual_user':
            user = 'hacker' + str(np.random.randint(1, 10))
            action = 'access'
            resource = 'system'
        elif anomaly_type == 'unusual_action':
            user = np.random.choice(users)
            action = 'delete'
            resource = 'system'
        else:
            user = np.random.choice(users)
            action = ' '.join(np.random.choice(actions, 3))
            resource = ' '.join(np.random.choice(resources, 2))

        log = f"User: {user} Action: {action} Resource: {resource}"
    Status: failed"

    return log

# Test real-time log monitor

```

```

def test_real_time_log_monitor():
    # Generate training logs (normal patterns)
    normal_logs = []
    users = ['alice', 'bob', 'charlie', 'david']
    normal_actions = ['login', 'logout', 'access', 'modify']
    resources = ['file', 'database', 'server']

    for _ in range(1000):
        user = np.random.choice(users)
        action = np.random.choice(normal_actions)
        resource = np.random.choice(resources)
        log = f"User: {user} Action: {action} Resource: {resource} Status: success"
        normal_logs.append(log)

    # Create monitor
    monitor = RealTimeLogMonitor(
        sequence_length=20,
        embedding_dim=16,
        window_size=500
    )

    # Train on normal logs
    monitor.train_on_normal_logs(normal_logs)

# Test with some anomalous logs
print("\nTesting anomaly detection...")

test_logs = [
    "User: alice Action: login Resource: server Status: success", #
Normal
    "User: hacker1 Action: access Resource: system Status: failed", #
Anomaly
    "User: bob Action: delete delete delete Resource: system system Status: failed", # Anomaly
    "User: charlie Action: modify Resource: file Status: success", #
Normal
]

for log in test_logs:
    result = monitor.process_log_stream(log)
    if result:
        print(f"\nLog: {log}")
        print(f"Anomaly: {result['is_anomaly']}, Score: {result['anomaly_score']:.2f}")

        if result['is_anomaly']:
            explanation = monitor.get_interpretable_explanation(log)
            print(explanation)

```

```
# Launch dashboard
print("\nLaunching real-time monitoring dashboard... ")
monitor.visualize_monitoring_dashboard(duration_seconds=15)

test_real_time_log_monitor()
```

---

**Task:** Create a framework that tests and improves autoencoder robustness to attacks.

```
class AdversarialRobustnessFramework:  
    """  
        Framework for testing and improving autoencoder robustness  
    """  
  
    def __init__(self, input_dim):  
        self.input_dim = input_dim  
        self.models = {}  
        self.attack_results = {}  
  
    def generate_adversarial_examples(self, model, X, method='fgsm',  
                                     epsilon=0.1):  
        """Generate adversarial examples using various methods"""  
  
        if method == 'fgsm':  
            # Fast Gradient Sign Method  
            X_adv = []  
  
            for x in X:  
                x_tensor = tf.Variable(x.reshape(1, -1), dtype=tf.float32)  
  
                with tf.GradientTape() as tape:  
                    reconstruction = model(x_tensor)  
                    loss = tf.reduce_mean(tf.square(x_tensor -  
reconstruction))  
  
                    # Get gradients  
                    gradients = tape.gradient(loss, x_tensor)  
  
                    # Create adversarial example  
                    x_adv = x_tensor + epsilon * tf.sign(gradients)  
                    X_adv.append(x_adv.numpy().reshape(-1))  
  
            return np.array(X_adv)  
  
        elif method == 'pgd':  
            # Projected Gradient Descent  
            X_adv = X.copy()  
  
            for iteration in range(10):  
                X_adv = self.generate_adversarial_examples(  
                    model, X_adv, method='fgsm', epsilon=epsilon/10  
                )  
                # Project back to valid range  
                X_adv = np.clip(X_adv, X - epsilon, X + epsilon)  
  
            return X_adv
```

```

    elif method == 'deepfool':
        # Simplified DeepFool
        X_adv = []

        for x in X:
            x_adv = x.copy()

            for _ in range(10):
                x_tensor = tf.Variable(x_adv.reshape(1, -1),
                                      dtype=tf.float32)

                with tf.GradientTape() as tape:
                    reconstruction = model(x_tensor)
                    loss = tf.reduce_mean(tf.square(x_tensor -
reconstruction))

                gradients = tape.gradient(loss,
x_tensor).numpy().reshape(-1)

                # Minimal perturbation
                perturbation = loss.numpy() * gradients /
(np.linalg.norm(gradients)**2 + 1e-8)
                x_adv += perturbation

            X_adv.append(x_adv)

        return np.array(X_adv)

def train_ensemble_with_augmentation(self, X_train, n_models=5):
    """Train ensemble with adversarial augmentation"""

    for i in range(n_models):
        print(f"\nTraining model {i+1}/{n_models}...")

        # Build autoencoder
        autoencoder = self.build_robust_autoencoder()

        # Generate adversarial training data
        X_adv_fgsm = self.generate_adversarial_examples(
            autoencoder, X_train[:100], method='fgsm', epsilon=0.1
        )
        X_adv_pgd = self.generate_adversarial_examples(
            autoencoder, X_train[:100], method='pgd', epsilon=0.05
        )

        # Combine training data
        X_augmented = np.vstack([X_train, X_adv_fgsm, X_adv_pgd])

        # Train with augmented data

```

```

        autoencoder.fit(
            X_augmented, X_augmented,
            epochs=30,
            batch_size=32,
            validation_split=0.2,
            verbose=0
        )

        self.models[f'robust_model_{i}'] = autoencoder

    return self.models

def build_robust_autoencoder(self):
    """Build autoencoder with robustness features"""

    # Input layer with noise
    inputs = layers.Input(shape=(self.input_dim,))

    # Add input noise for robustness
    noisy_inputs = layers.GaussianNoise(0.1)(inputs)

    # Encoder with gradient regularization
    x = layers.Dense(128, activation='relu')(noisy_inputs)
    x = layers.BatchNormalization()(x)
    x = layers.Dropout(0.3)(x)

    x = layers.Dense(64, activation='relu')(x)
    x = layers.BatchNormalization()(x)
    x = layers.Dropout(0.3)(x)

    encoded = layers.Dense(32, activation='relu')(x)

    # Decoder
    x = layers.Dense(64, activation='relu')(encoded)
    x = layers.BatchNormalization()(x)
    x = layers.Dropout(0.3)(x)

    x = layers.Dense(128, activation='relu')(x)
    x = layers.BatchNormalization()(x)

    decoded = layers.Dense(self.input_dim, activation='linear')(x)

    # Model
    autoencoder = keras.Model(inputs, decoded)

    # Custom loss with gradient penalty
    def robust_loss(y_true, y_pred):
        # Reconstruction loss
        recon_loss = tf.reduce_mean(tf.square(y_true - y_pred))

```

```
# Gradient penalty for smoothness
with tf.GradientTape() as tape:
    tape.watch(y_true)
    pred = autoencoder(y_true)

    gradients = tape.gradient(pred, y_true)
    gradient_penalty = tf.reduce_mean(tf.square(gradients))

return recon_loss + 0.1 * gradient_penalty

autoencoder.compile(optimizer='adam', loss='mse')

return autoencoder

def test_robustness(self, X_test, y_test):
    """Test model robustness to various attacks"""

attack_methods = {
    'fgsm_weak': {'method': 'fgsm', 'epsilon': 0.05},
    'fgsm_strong': {'method': 'fgsm', 'epsilon': 0.2},
    'pgd': {'method': 'pgd', 'epsilon': 0.1},
    'deepfool': {'method': 'deepfool', 'epsilon': None}
}

for model_name, model in self.models.items():
    print(f"\nTesting {model_name}...")
    self.attack_results[model_name] = {}

    for attack_name, attack_params in attack_methods.items():
        # Generate adversarial examples
        X_adv = self.generate_adversarial_examples(
            model, X_test,
            method=attack_params['method'],
            epsilon=attack_params.get('epsilon', 0.1)
        )

        # Test detection
        X_clean_recon = model.predict(X_test, verbose=0)
        X_adv_recon = model.predict(X_adv, verbose=0)

        clean_errors = np.mean((X_test - X_clean_recon)**2,
axis=1)
        adv_errors = np.mean((X_adv - X_adv_recon)**2, axis=1)

        # Calculate threshold
        threshold = np.percentile(clean_errors, 95)

        # Detection rates
        clean_detected = np.mean(clean_errors > threshold)
        adv_detected = np.mean(adv_errors > threshold)
```

```

        self.attack_results[model_name][attack_name] = {
            'clean_detected': clean_detected,
            'adv_detected': adv_detected,
            'threshold': threshold,
            'perturbation_norm': np.mean(np.linalg.norm(X_adv -
X_test, axis=1))
        }

    def implement_defense_mechanisms(self):
        """Implement various defense mechanisms"""

        defenses = {}

        # 1. Input validation
        def input_validation(X, max_norm=10.0):
            """Reject inputs with suspiciously large norms"""
            norms = np.linalg.norm(X, axis=1)
            valid_mask = norms < max_norm
            return X[valid_mask], valid_mask

        # 2. Ensemble voting
        def ensemble_voting(X, threshold_percentile=95):
            """Use ensemble voting for robust detection"""
            all_errors = []

            for model in self.models.values():
                reconstruction = model.predict(X, verbose=0)
                errors = np.mean((X - reconstruction)**2, axis=1)
                all_errors.append(errors)

            # Median error across models
            median_errors = np.median(all_errors, axis=0)
            threshold = np.percentile(median_errors, threshold_percentile)

            return median_errors > threshold, median_errors

        # 3. Randomized smoothing
        def randomized_smoothing(X, model, n_samples=10, sigma=0.1):
            """Add noise and average predictions"""
            predictions = []

            for _ in range(n_samples):
                X_noisy = X + np.random.normal(0, sigma, X.shape)
                pred = model.predict(X_noisy, verbose=0)
                predictions.append(pred)

            return np.mean(predictions, axis=0)

        defenses['input_validation'] = input_validation

```

```

        defenses['ensemble_voting'] = ensemble_voting
        defenses['randomized_smoothing'] = randomized_smoothing

    return defenses

def measure_stability(self, X_test):
    """Measure model stability under perturbations"""

    stability_results = {}

    for model_name, model in self.models.items():
        stabilities = []

        # Test with different noise levels
        noise_levels = np.linspace(0, 0.5, 10)

        for noise_level in noise_levels:
            # Add noise
            X_noisy = X_test + np.random.normal(0, noise_level,
X_test.shape)

            # Get reconstructions
            X_clean_recon = model.predict(X_test, verbose=0)
            X_noisy_recon = model.predict(X_noisy, verbose=0)

            # Measure stability
            stability = 1 - np.mean(
                np.linalg.norm(X_clean_recon - X_noisy_recon, axis=1)
/
                (np.linalg.norm(X_clean_recon, axis=1) + 1e-8)
            )

            stabilities.append(stability)

        stability_results[model_name] = {
            'noise_levels': noise_levels,
            'stabilities': stabilities,
            'avg_stability': np.mean(stabilities)
        }

    return stability_results

def visualize_robustness_analysis(self, X_test, y_test):
    """Comprehensive visualization of robustness analysis"""

    fig, axes = plt.subplots(2, 3, figsize=(15, 10))

    # 1. Attack success rates
    ax = axes[0, 0]

```

```

    attack_names =
list(next(iter(self.attack_results.values()))).keys())
model_names = list(self.models.keys())

# Create heatmap data
success_rates = np.zeros((len(model_names), len(attack_names)))

for i, model_name in enumerate(model_names):
    for j, attack_name in enumerate(attack_names):
        result = self.attack_results[model_name][attack_name]
        # Success rate = adversarial examples NOT detected
        success_rates[i, j] = 1 - result['adv_detected']

im = ax.imshow(success_rates, cmap='RdYlGn_r', vmin=0, vmax=1)
ax.set_xticks(range(len(attack_names)))
ax.set_yticks(range(len(model_names)))
ax.set_xticklabels(attack_names, rotation=45)
ax.set_yticklabels(model_names)
ax.set_title('Attack Success Rates')
plt.colorbar(im, ax=ax)

# Add values
for i in range(len(model_names)):
    for j in range(len(attack_names)):
        ax.text(j, i, f'{success_rates[i, j]:.2f}', ha='center', va='center') 462

# 2. Perturbation sizes
ax = axes[0, 1]

for attack_name in attack_names:
    perturbation_norms = []
    for model_name in model_names:
        norm = self.attack_results[model_name][attack_name]
        ['perturbation_norm']
        perturbation_norms.append(norm)

    ax.plot(model_names, perturbation_norms, 'o-', label=attack_name)

    ax.set_ylabel('Average Perturbation Norm')
    ax.set_title('Perturbation Sizes by Attack')
    ax.legend()
    ax.grid(True, alpha=0.3)
    ax.tick_params(axis='x', rotation=45)

# 3. ROC curves
ax = axes[0, 2]

for model_name, model in self.models.items():

```

```
# Get predictions
X_clean_recon = model.predict(X_test, verbose=0)
errors = np.mean((X_test - X_clean_recon)**2, axis=1)

# Calculate ROC
from sklearn.metrics import roc_curve, auc
fpr, tpr, _ = roc_curve(y_test, errors)
roc_auc = auc(fpr, tpr)

ax.plot(fpr, tpr, label=f'{model_name} (AUC: {roc_auc:.3f})')

ax.plot([0, 1], [0, 1], 'k--')
ax.set_xlabel('False Positive Rate')
ax.set_ylabel('True Positive Rate')
ax.set_title('ROC Curves')
ax.legend()
ax.grid(True, alpha=0.3)

# 4. Stability analysis
ax = axes[1, 0]

stability_results = self.measure_stability(X_test[:100])

for model_name, results in stability_results.items():
    ax.plot(results['noise_levels'], results['stabilities'],
            'o-', label=f'{model_name} (avg: {results["avg_stability"]:.3f})')

ax.set_xlabel('Noise Level')
ax.set_ylabel('Stability Score')
ax.set_title('Model Stability Under Noise')
ax.legend()
ax.grid(True, alpha=0.3)

# 5. Ensemble performance
ax = axes[1, 1]

# Test ensemble
defenses = self.implement_defense_mechanisms()
ensemble_predictions, ensemble_scores =
defenses['ensemble_voting'](X_test)

# Individual vs ensemble
individual_accs = []
for model in self.models.values():
    recon = model.predict(X_test, verbose=0)
    errors = np.mean((X_test - recon)**2, axis=1)
    threshold = np.percentile(errors[y_test == 0], 95)
    predictions = errors > threshold
    accuracy = np.mean(predictions == y_test)
```

```

        individual_accs.append(accuracy)

ensemble_acc = np.mean(ensemble_predictions == y_test)

models = list(self.models.keys()) + ['Ensemble']
accuracies = individual_accs + [ensemble_acc]

bars = ax.bar(models, accuracies)
bars[-1].set_color('gold')

ax.set_ylabel('Detection Accuracy')
ax.set_title('Individual vs Ensemble Performance')
ax.tick_params(axis='x', rotation=45)
ax.grid(True, alpha=0.3, axis='y')

# 6. Defense effectiveness
ax = axes[1, 2]

defense_names = ['No Defense', 'Input Validation', 'Ensemble',
'Smoothing']
defense_scores = [0.5, 0.65, 0.8, 0.75] # Simulated scores

bars = ax.bar(defense_names, defense_scores, color=['red',
'orange', 'yellow', 'green'])
ax.set_ylabel('Robustness Score')
ax.set_title('Defense Mechanism Effectiveness')
ax.tick_params(axis='x', rotation=45)
ax.grid(True, alpha=0.3, axis='y')

plt.suptitle('Adversarial Robustness Analysis', fontsize=16)
plt.tight_layout()
plt.show()

# Test adversarial robustness
def test_adversarial_robustness():
    # Generate synthetic data
    np.random.seed(42)
    n_samples = 2000
    n_features = 50

    # Normal data
    X_normal = np.random.randn(n_samples, n_features) * 0.5

    # Anomalous data
    X_anomaly = np.random.randn(200, n_features) * 1.5 + 1

    # Combine
    X = np.vstack([X_normal, X_anomaly])
    y = np.array([0] * n_samples + [1] * 200)

```

```
# Shuffle
indices = np.random.permutation(len(X))
X = X[indices]
y = y[indices]

# Split
split = int(0.8 * len(X))
X_train = X[:split]
X_test = X[split:]
y_test = y[split:]

# Use only normal data for training
X_train_normal = X_train[y[:split] == 0]

# Create framework
framework = AdversarialRobustnessFramework(input_dim=n_features)

# Train ensemble with adversarial augmentation
framework.train_ensemble_with_augmentation(X_train_normal, n_models=3)

# Test robustness
framework.test_robustness(X_test, y_test)

# Visualize results
framework.visualize_robustness_analysis(X_test, y_test)

# Print summary
print("\n==== Robustness Summary ====")
for model_name, attacks in framework.attack_results.items():
    print(f"\n{model_name}:")
    for attack_name, results in attacks.items():
        print(f"  {attack_name}: Detection rate = {results['adv_detected']:.2%}")

test_adversarial_robustness()
```

# Exercise 1: Multi-Modal Security Model

**Task:** Build a model that combines network traffic, system logs, and user behavior.

```
class MultiModalSecurityModel:  
    """  
        Multi-modal model combining different data types for threat detection  
    """  
  
    def __init__(self):  
        self.model = None  
        self.preprocessors = {}  
        self.attention_weights = None  
  
    def build_multi_modal_model(self, network_shape, log_shape,  
behavior_shape):  
        """Build model with multiple inputs and attention mechanisms"""\n  
  
        # Network traffic branch (CNN-based)  
        network_input = layers.Input(shape=network_shape,  
name='network_input')  
        network_branch = layers.Conv1D(64, 3, activation='relu')(network_input)  
        network_branch = layers.MaxPooling1D(2)(network_branch)  
        network_branch = layers.Conv1D(32, 3, activation='relu')(network_branch)  
        network_branch = layers.GlobalMaxPooling1D()(network_branch)  
        network_branch = layers.Dense(64, activation='relu')(network_branch)  
  
        # System log branch (RNN-based)  
        log_input = layers.Input(shape=log_shape, name='log_input')  
        log_branch = layers.LSTM(64, return_sequences=True)(log_input)  
        log_branch = layers.LSTM(32)(log_branch)  
        log_branch = layers.Dense(64, activation='relu')(log_branch)  
  
        # User behavior branch (Dense)  
        behavior_input = layers.Input(shape=behavior_shape,  
name='behavior_input')  
        behavior_branch = layers.Dense(128, activation='relu')(behavior_input)  
        behavior_branch = layers.BatchNormalization()(behavior_branch)  
        behavior_branch = layers.Dropout(0.3)(behavior_branch)  
        behavior_branch = layers.Dense(64, activation='relu')(behavior_branch)  
  
        # Attention mechanism between modalities  
        # Self-attention for each branch  
        network_attention = layers.MultiHeadAttention(  
466
```

```

        num_heads=4, key_dim=64
    )(network_branch[..., tf.newaxis], network_branch[...,
tf.newaxis])
    network_attention = layers.Flatten()(network_attention)

    log_attention = layers.MultiHeadAttention(
        num_heads=4, key_dim=64
    )(log_branch[..., tf.newaxis], log_branch[..., tf.newaxis])
    log_attention = layers.Flatten()(log_attention)

    behavior_attention = layers.MultiHeadAttention(
        num_heads=4, key_dim=64
    )(behavior_branch[..., tf.newaxis], behavior_branch[...,
tf.newaxis])
    behavior_attention = layers.Flatten()(behavior_attention)

    # Cross-modal attention
    all_features = layers.concatenate([
        network_attention,
        log_attention,
        behavior_attention
    ])

    # Modality importance weights (learnable)
    attention_weights = layers.Dense(3, activation='softmax',
                                    name='modality_weights')
(all_features)

    # Apply attention weights
    weighted_network = layers.Multiply()([network_branch,
attention_weights[:, 0:1]])
    weighted_log = layers.Multiply()([log_branch, attention_weights[:, 1:2]])
    weighted_behavior = layers.Multiply()([behavior_branch,
attention_weights[:, 2:3]])

    # Combine weighted features
    combined = layers.concatenate([
        weighted_network,
        weighted_log,
        weighted_behavior
    ])

    # Final classification layers
    x = layers.Dense(128, activation='relu')(combined)
    x = layers.BatchNormalization()(x)
    x = layers.Dropout(0.4)(x)
    x = layers.Dense(64, activation='relu')(x)

    # Multiple outputs

```

```

        threat_level = layers.Dense(1, activation='sigmoid',
                                     name='threat_level')(x)
        attack_type = layers.Dense(5, activation='softmax',
                                   name='attack_type')(x)

    # Model
    model = keras.Model(
        inputs=[network_input, log_input, behavior_input],
        outputs=[threat_level, attack_type]
    )

    # Attention extractor
    self.attention_extractor = keras.Model(
        inputs=[network_input, log_input, behavior_input],
        outputs=attention_weights
    )

    return model

def create_preprocessors(self):
    """Create preprocessing pipelines for each modality"""

    # Network traffic preprocessor
    class NetworkPreprocessor:
        def __init__(self):
            self.scaler = StandardScaler()

        def fit(self, data):
            # Flatten temporal data for scaling
            n_samples, n_time, n_features = data.shape
            data_flat = data.reshape(-1, n_features)
            self.scaler.fit(data_flat)
            return self

        def transform(self, data):
            n_samples, n_time, n_features = data.shape
            data_flat = data.reshape(-1, n_features)
            scaled = self.scaler.transform(data_flat)
            return scaled.reshape(n_samples, n_time, n_features)

    # Log preprocessor
    class LogPreprocessor:
        def __init__(self, max_features=1000):
            self.tokenizer = None
            self.max_features = max_features

        def fit(self, logs):
            # Simple tokenization
            from sklearn.feature_extraction.text import
CountVectorizer

```

```

        self.tokenizer =
CountVectorizer(max_features=self.max_features)

        # Flatten logs
        all_logs = [' '.join(log_seq) for log_seq in logs]
        self.tokenizer.fit(all_logs)
        return self

    def transform(self, logs):
        # Convert to sequences of token indices
        sequences = []
        for log_seq in logs:
            seq = []
            for log in log_seq:
                tokens = self.tokenizer.transform([log]).toarray()
[0]
                seq.append(tokens)
            sequences.append(seq)
        return np.array(sequences)

# Behavior preprocessor
class BehaviorPreprocessor:
    def __init__(self):
        self.scaler = StandardScaler()
        self.feature_names = None

    def fit(self, data):
        self.scaler.fit(data)
        return self

    def transform(self, data):
        return self.scaler.transform(data)

self.preprocessors = {
    'network': NetworkPreprocessor(),
    'log': LogPreprocessor(),
    'behavior': BehaviorPreprocessor()
}

return self.preprocessors

def train_model(self, X_network, X_log, X_behavior, y_threat,
y_attack,
                    epochs=50, batch_size=32):
    """Train the multi-modal model"""

    # Compile model
    self.model.compile(
        optimizer='adam',
        loss={

```

```

        'threat_level': 'binary_crossentropy',
        'attack_type': 'sparse_categorical_crossentropy'
    },
    loss_weights={
        'threat_level': 1.0,
        'attack_type': 0.5
    },
    metrics={
        'threat_level': ['accuracy', tf.keras.metrics.AUC()],
        'attack_type': 'accuracy'
    }
)

# Custom callback to track attention weights
class AttentionTracker(keras.callbacks.Callback):
    def __init__(self, attention_model, validation_data):
        self.attention_model = attention_model
        self.validation_data = validation_data
        self.attention_history = []

    def on_epoch_end(self, epoch, logs=None):
        # Get attention weights on validation data
        val_attention = self.attention_model.predict(
            self.validation_data[:3], verbose=0
        )
        avg_attention = np.mean(val_attention, axis=0)
        self.attention_history.append(avg_attention)

    # Prepare validation data
    val_split = int(0.8 * len(X_network))
    X_network_train, X_network_val = X_network[:val_split],
X_network[val_split:]
    X_log_train, X_log_val = X_log[:val_split], X_log[val_split:]
    X_behavior_train, X_behavior_val = X_behavior[:val_split],
X_behavior[val_split:]
    y_threat_train, y_threat_val = y_threat[:val_split],
y_threat[val_split:]
    y_attack_train, y_attack_val = y_attack[:val_split],
y_attack[val_split:]

    attention_tracker = AttentionTracker(
        self.attention_extractor,
        [X_network_val[:100], X_log_val[:100], X_behavior_val[:100]]
    )

    # Train
    history = self.model.fit(
        [X_network_train, X_log_train, X_behavior_train],
        {'threat_level': y_threat_train, 'attack_type':
y_attack_train},

```

```

        validation_data=(
            [X_network_val, X_log_val, X_behavior_val],
            {'threat_level': y_threat_val, 'attack_type':
y_attack_val}
        ),
        epochs=epochs,
        batch_size=batch_size,
        callbacks=[attention_tracker],
        verbose=1
    )

    self.attention_weights = attention_tracker.attention_history

    return history

def get_interpretable_features(self, X_network, X_log, X_behavior):
    """Extract interpretable feature importance"""

    # Get predictions and attention weights
    predictions = self.model.predict([X_network, X_log, X_behavior])
    attention = self.attention_extractor.predict([X_network, X_log,
X_behavior])

    # Create intermediate models for each branch
    network_model = keras.Model(
        inputs=self.model.get_layer('network_input').input,
        outputs=self.model.get_layer('network_input').output
    )

    # Feature importance analysis
    feature_importance = {
        'modality_importance': np.mean(attention, axis=0),
        'network_activation':
np.mean(np.abs(network_model.predict(X_network))),
        'predictions': predictions
    }

    return feature_importance

def visualize_multi_modal_analysis(self, X_network, X_log, X_behavior,
                                    y_threat, y_attack):
    """Comprehensive visualization of multi-modal model"""

    fig, axes = plt.subplots(2, 3, figsize=(15, 10))

    # 1. Modality importance over training
    ax = axes[0, 0]
    if self.attention_weights:
        attention_array = np.array(self.attention_weights)
        epochs = range(len(attention_array))

```

```

        ax.plot(epochs, attention_array[:, 0], 'b-', label='Network',
linewidth=2)
        ax.plot(epochs, attention_array[:, 1], 'g-', label='Logs',
linewidth=2)
        ax.plot(epochs, attention_array[:, 2], 'r-', label='Behavior',
linewidth=2)

    ax.set_xlabel('Epoch')
    ax.set_ylabel('Attention Weight')
    ax.set_title('Modality Importance During Training')
    ax.legend()
    ax.grid(True, alpha=0.3)

# 2. Prediction performance
ax = axes[0, 1]

# Get predictions
threat_pred, attack_pred = self.model.predict(
    [X_network, X_log, X_behavior], verbose=0
)

# ROC curve for threat detection
from sklearn.metrics import roc_curve, auc
fpr, tpr, _ = roc_curve(y_threat, threat_pred)
roc_auc = auc(fpr, tpr) 472

    472

ax.plot(fpr, tpr, 'b-', linewidth=2,
        label=f'Threat Detection (AUC: {roc_auc:.3f})')
ax.plot([0, 1], [0, 1], 'k--')
ax.set_xlabel('False Positive Rate')
ax.set_ylabel('True Positive Rate')
ax.set_title('Threat Detection Performance')
ax.legend()
ax.grid(True, alpha=0.3)

# 3. Attack type confusion matrix
ax = axes[0, 2]

from sklearn.metrics import confusion_matrix
attack_pred_classes = np.argmax(attack_pred, axis=1)
cm = confusion_matrix(y_attack, attack_pred_classes)

im = ax.imshow(cm, cmap='Blues')
ax.set_xlabel('Predicted')
ax.set_ylabel('Actual')
ax.set_title('Attack Type Classification')
plt.colorbar(im, ax=ax)

# Add numbers

```

```

        for i in range(cm.shape[0]):
            for j in range(cm.shape[1]):
                ax.text(j, i, str(cm[i, j]), ha='center', va='center')

    # 4. Feature correlation heatmap
    ax = axes[1, 0]

    # Get attention weights for samples
    sample_attention = self.attention_extractor.predict(
        [X_network[:100], X_log[:100], X_behavior[:100]], verbose=0
    )

    # Correlate with threat levels
    correlations = np.zeros((3, 3))
    modalities = ['Network', 'Logs', 'Behavior']

    for i in range(3):
        for j in range(3):
            correlations[i, j] = np.corrcoef(
                sample_attention[:, i],
                sample_attention[:, j]
            )[0, 1]

    im = ax.imshow(correlations, cmap='RdBu', vmin=-1, vmax=1)
    ax.set_xticks(range(3))
    ax.set_yticks(range(3))
    ax.set_xticklabels(modalities)
    ax.set_yticklabels(modalities)
    ax.set_title('Modality Correlation')
    plt.colorbar(im, ax=ax)

    # Add values
    for i in range(3):
        for j in range(3):
            ax.text(j, i, f'{correlations[i, j]:.2f}',
                    ha='center', va='center')

    # 5. Sample predictions with attention
    ax = axes[1, 1]

    # Show a few samples
    n_samples = 5
    sample_indices = np.random.choice(len(X_network), n_samples,
replace=False)

    sample_data = []
    for idx in sample_indices:
        attention = self.attention_extractor.predict(
            [X_network[idx:idx+1], X_log[idx:idx+1],
             X_behavior[idx:idx+1]],
```

```

        verbose=0
    )[0]
    threat = threat_pred[idx][0]
    attack = np.argmax(attack_pred[idx])
    sample_data.append([threat, attack] + list(attention))

sample_data = np.array(sample_data)

# Create bar chart
x = np.arange(n_samples)
width = 0.2

    ax.bar(x - width, sample_data[:, 2], width, label='Network',
alpha=0.8)
    ax.bar(x, sample_data[:, 3], width, label='Logs', alpha=0.8)
    ax.bar(x + width, sample_data[:, 4], width, label='Behavior',
alpha=0.8)

    ax.set_xlabel('Sample')
    ax.set_ylabel('Attention Weight')
    ax.set_title('Per-Sample Modality Importance')
    ax.legend()
    ax.grid(True, alpha=0.3, axis='y')

# 6. Performance metrics
ax = axes[1, 2]

metrics_text = "Multi-Modal Model Performance\n\n"

# Threat detection metrics
threat_acc = np.mean((threat_pred > 0.5).astype(int).flatten() ==
y_threat)
    metrics_text += f"Threat Detection Accuracy: {threat_acc:.3f}\n"
    metrics_text += f"Threat Detection AUC: {roc_auc:.3f}\n\n"

# Attack classification metrics
attack_acc = np.mean(attack_pred_classes == y_attack)
    metrics_text += f"Attack Type Accuracy: {attack_acc:.3f}\n\n"

# Modality importance
avg_attention = np.mean(sample_attention, axis=0)
metrics_text += "Average Modality Importance:\n"
metrics_text += f" Network: {avg_attention[0]:.3f}\n"
metrics_text += f" Logs: {avg_attention[1]:.3f}\n"
metrics_text += f" Behavior: {avg_attention[2]:.3f}\n"

ax.text(0.1, 0.9, metrics_text, transform=ax.transAxes,
        fontsize=10, verticalalignment='top',
        fontfamily='monospace')
ax.axis('off')

```

```
plt.suptitle('Multi-Modal Security Model Analysis', fontsize=16)
plt.tight_layout()
plt.show()

# Test multi-modal security model
def test_multi_modal_model():
    # Generate synthetic multi-modal data
    np.random.seed(42)
    n_samples = 2000

    # Network traffic data (time series)
    X_network = np.random.randn(n_samples, 50, 10) # 50 time steps, 10
features

    # Log data (sequences)
    X_log = np.random.randn(n_samples, 20, 100) # 20 logs, 100 features
each

    # User behavior data (tabular)
    X_behavior = np.random.randn(n_samples, 30) # 30 behavioral features

    # Labels
    y_threat = np.random.randint(0, 2, n_samples) # Binary threat level
    y_attack = np.random.randint(0, 5, n_samples) # 5 attack types 475

    # Add patterns based on labels
    for i in range(n_samples):
        if y_threat[i] == 1:
            # Add anomaly patterns
            X_network[i] += np.random.randn(50, 10) * 0.5
            X_log[i, :, :20] += 1.0
            X_behavior[i, :10] += 0.5

            # Attack-specific patterns
            X_network[i, :, y_attack[i]] += 0.3
            X_behavior[i, y_attack[i]*5:(y_attack[i]+1)*5] += 0.5

    # Create and train model
    mm_model = MultiModalSecurityModel()

    # Build model
    model = mm_model.build_multi_modal_model(
        network_shape=(50, 10),
        log_shape=(20, 100),
        behavior_shape=(30,))
    mm_model.model = model

    # Create preprocessors
```

```
mm_model.create_preprocessors()

# Train
print("Training multi-modal security model...")
history = mm_model.train_model(
    X_network, X_log, X_behavior,
    y_threat, y_attack,
    epochs=20,
    batch_size=32
)

# Visualize analysis
mm_model.visualize_multi_modal_analysis(
    X_network[-200:], X_log[-200:], X_behavior[-200:],
    y_threat[-200:], y_attack[-200:]
)

test_multi_modal_model()
```

**Task:** Create an augmentation system that learns which augmentations improve performance.

```
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

class AdaptiveDataAugmentation:
    """
    Data augmentation system that learns optimal augmentation strategies
    """

    def __init__(self, augmentation_space):
        self.augmentation_space = augmentation_space
        self.augmentation_history = []
        self.performance_history = []
        self.learned_policy = None

    def create_augmentation_functions(self):
        """Create various augmentation functions"""

        augmentations = {
            'noise': lambda x, strength: x + np.random.normal(0, strength,
x.shape),
            'scaling': lambda x, strength: x * (1 + np.random.uniform(-
strength, strength)),
            'rotation': lambda x, strength: self._rotate_features(x,
strength),
            'dropout': lambda x, strength: x * (np.random.random(x.shape)
> strength),
            'mixup': lambda x1, x2, strength: strength * x1 + (1 -
strength) * x2,
            'cutmix': lambda x1, x2, strength: self._cutmix(x1, x2,
strength),
            'adversarial': lambda x, strength: x + strength *
np.sign(np.random.randn(*x.shape))
        }

        return augmentations

    def _rotate_features(self, x, angle):
        """Rotate features (for 2D data)"""
        if len(x.shape) == 2:
            # Simple rotation in feature space
            theta = np.radians(angle)
            rotation_matrix = np.array([

```

```

        [np.cos(theta), -np.sin(theta)],
        [np.sin(theta), np.cos(theta)]
    ])

    # Apply to first two features
    rotated = x.copy()
    rotated[:, :2] = x[:, :2] @ rotation_matrix.T
    return rotated
return x

def _cutmix(self, x1, x2, lam):
    """CutMix augmentation"""
    if len(x1.shape) == 1:
        # 1D CutMix
        cut_point = int(len(x1) * lam)
        mixed = x1.copy()
        mixed[cut_point:] = x2[cut_point:]
        return mixed
    elif len(x1.shape) == 2:
        # 2D CutMix
        h, w = x1.shape
        cut_h = int(h * lam)
        mixed = x1.copy()
        mixed[cut_h:, :] = x2[cut_h:, :]
        return mixed
    return x1

def learn_augmentation_policy(self, X_train, y_train, X_val, y_val,
                             base_model_fn, n_trials=50):
    """Learn which augmentations work best"""

    augmentation_fns = self.create_augmentation_functions()

    for trial in range(n_trials):
        print(f"\nTrial {trial + 1}/{n_trials}")

        # Sample augmentation strategy
        strategy = {}
        for aug_name in self.augmentation_space.keys():
            if np.random.random() < 0.5: # 50% chance to use each
                augmentation
                    strength = np.random.uniform(
                        self.augmentation_space[aug_name]['min'],
                        self.augmentation_space[aug_name]['max']
                    )
                    strategy[aug_name] = strength

        # Apply augmentations
        X_augmented = self._apply_augmentation_strategy(
            X_train, y_train, strategy, augmentation_fns

```

```

        )

# Train model with augmented data
model = base_model_fn()
model.fit(X_augmented, y_train, epochs=10, verbose=0)

# Evaluate
val_loss, val_acc = model.evaluate(X_val, y_val, verbose=0)

# Store results
self.augmentation_history.append(strategy)
self.performance_history.append(val_acc)

print(f"Strategy: {strategy}")
print(f"Validation accuracy: {val_acc:.4f}")

# Learn policy from results
self._learn_policy()

return self.learned_policy

def _apply_augmentation_strategy(self, X, y, strategy,
augmentation_fns):
    """Apply augmentation strategy to data"""
    X_augmented = X.copy()

    for aug_name, strength in strategy.items():
        if aug_name in ['mixup', 'cutmix']:
            # Pairwise augmentations
            indices = np.random.permutation(len(X))
            X_pairs = X[indices]

            for i in range(len(X)):
                X_augmented[i] = augmentation_fns[aug_name](
                    X[i], X_pairs[i], strength
                )
        else:
            # Single sample augmentations
            for i in range(len(X)):
                X_augmented[i] = augmentation_fns[aug_name](X[i],
strength)

    return X_augmented

def _learn_policy(self):
    """Learn optimal policy from history"""
    # Find top performing strategies
    performances = np.array(self.performance_history)
    top_indices = np.argsort(performances)[-10:] # Top 10

```

```

# Average top strategies
learned_policy = {}

for aug_name in self.augmentation_space.keys():
    strengths = []
    for idx in top_indices:
        strategy = self.augmentation_history[idx]
        if aug_name in strategy:
            strengths.append(strategy[aug_name])

    if strengths:
        learned_policy[aug_name] = {
            'use_probability': len(strengths) / len(top_indices),
            'strength': np.mean(strengths)
        }

self.learned_policy = learned_policy

def generate_synthetic_data(self, X, y, n_synthetic):
    """Generate synthetic training data"""
    augmentation_fns = self.create_augmentation_functions()
    X_synthetic = []
    y_synthetic = []

    for _ in range(n_synthetic):
        # Random sample
        idx = np.random.randint(len(X))
        x_sample = X[idx]
        y_sample = y[idx]

        # Apply learned policy
        x_aug = x_sample.copy()

        if self.learned_policy:
            for aug_name, policy in self.learned_policy.items():
                if np.random.random() < policy['use_probability']:
                    if aug_name in ['mixup', 'cutmix']:
                        # Need another sample
                        idx2 = np.random.randint(len(X))
                        x_aug = augmentation_fns[aug_name](
                            x_aug, X[idx2], policy['strength'])
                    else:
                        x_aug = augmentation_fns[aug_name](
                            x_aug, policy['strength'])

        X_synthetic.append(x_aug)
        y_synthetic.append(y_sample)

```

```

        return np.array(X_synthetic), np.array(y_synthetic)

def apply_adversarial_perturbation(self, X, model, epsilon=0.1):
    """Apply adversarial perturbations for robustness"""
    X_adv = []

    for x in X:
        x_tensor = tf.Variable(x.reshape(1, -1), dtype=tf.float32)

        with tf.GradientTape() as tape:
            prediction = model(x_tensor)
            loss = tf.reduce_mean(prediction)

        # Get gradients
        gradients = tape.gradient(loss, x_tensor)

        # FGSM perturbation
        perturbation = epsilon * tf.sign(gradients)
        x_adv = x_tensor + perturbation

        X_adv.append(x_adv.numpy().reshape(-1))

    return np.array(X_adv)

def balance_augmentation_strength(self, X, y):
    """Balance augmentation strength with label preservation"""
    # Test different augmentation strengths
    strengths = np.linspace(0, 1, 10)
    preservation_scores = []

    augmentation_fns = self.create_augmentation_functions()

    for strength in strengths:
        # Apply augmentation
        strategy = {aug: strength for aug in
self.augmentation_space.keys()}
        X_aug = self._apply_augmentation_strategy(
            X[:100], y[:100], strategy, augmentation_fns
        )

        # Measure label preservation (using nearest neighbor)
        from sklearn.neighbors import KNeighborsClassifier
        knn = KNeighborsClassifier(n_neighbors=5)
        knn.fit(X, y)

        # Predict labels for augmented data
        y_pred = knn.predict(X_aug)
        preservation = np.mean(y_pred == y[:100])
        preservation_scores.append(preservation)

```

```

        return strengths, preservation_scores

    def adapt_during_training(self, model, X_train, y_train, X_val,
y_val):
        """Adapt augmentation strategy during training"""
        augmentation_fns = self.create_augmentation_functions()
        adapted_history = []

        for epoch in range(20):
            # Evaluate current performance
            val_loss, val_acc = model.evaluate(X_val, y_val, verbose=0)

            # Adapt strategy based on performance
            if epoch > 0 and val_acc < adapted_history[-1]['val_acc']:
                # Performance dropped, reduce augmentation
                for aug_name in self.learned_policy:
                    self.learned_policy[aug_name]['strength'] *= 0.9
            elif epoch > 5 and val_acc > np.mean([h['val_acc'] for h in
adapted_history[-5:]]):
                # Performance improving, increase augmentation
                for aug_name in self.learned_policy:
                    self.learned_policy[aug_name]['strength'] *= 1.1

            # Apply current augmentation
            X_aug = self._apply_augmentation_strategy(
                X_train, y_train,
                {k: v['strength'] for k, v in
self.learned_policy.items()},
                augmentation_fns
            )

            # Train epoch
            history = model.fit(X_aug, y_train, epochs=1, verbose=0)

            adapted_history.append({
                'epoch': epoch,
                'val_acc': val_acc,
                'augmentation_strength': np.mean([v['strength']
for v in
self.learned_policy.values()])
            })

        return adapted_history

    def visualize_augmentation_analysis(self, X_test, y_test):
        """Visualize augmentation system analysis"""

        fig, axes = plt.subplots(2, 3, figsize=(15, 10))

        # 1. Augmentation performance history

```

```

ax = axes[0, 0]

if self.performance_history:
    ax.scatter(range(len(self.performance_history)),
               self.performance_history, alpha=0.6)

# Mark top performers
top_indices = np.argsort(self.performance_history)[-5:]
ax.scatter(top_indices,
           [self.performance_history[i] for i in top_indices],
           color='red', s=100, label='Top 5')

ax.set_xlabel('Trial')
ax.set_ylabel('Validation Accuracy')
ax.set_title('Augmentation Strategy Performance')
ax.legend()
ax.grid(True, alpha=0.3)

# 2. Learned policy
ax = axes[0, 1]

if self.learned_policy:
    aug_names = list(self.learned_policy.keys())
    use_probs = [v['use_probability'] for v in
    self.learned_policy.values()]
    strengths = [v['strength'] for v in
    self.learned_policy.values()]

    x = np.arange(len(aug_names))
    width = 0.35

    ax.bar(x - width/2, use_probs, width, label='Use Probability',
alpha=0.8)
    ax.bar(x + width/2, strengths, width, label='Strength',
alpha=0.8)

    ax.set_xlabel('Augmentation Type')
    ax.set_ylabel('Value')
    ax.set_title('Learned Augmentation Policy')
    ax.set_xticks(x)
    ax.set_xticklabels(aug_names, rotation=45)
    ax.legend()
    ax.grid(True, alpha=0.3, axis='y')

# 3. Label preservation vs strength
ax = axes[0, 2]

strengths, preservation =
self.balance_augmentation_strength(X_test, y_test)

```

483

```

        ax.plot(strengths, preservation, 'b-', linewidth=2)
        ax.axhline(y=0.9, color='red', linestyle='--',
                    label='90% preservation threshold')
        ax.set_xlabel('Augmentation Strength')
        ax.set_ylabel('Label Preservation Rate')
        ax.set_title('Augmentation vs Label Preservation')
        ax.legend()
        ax.grid(True, alpha=0.3)

# 4. Augmentation examples
ax = axes[1, 0]

# Show original vs augmented samples
sample_idx = 0
original = X_test[sample_idx]

augmentation_fns = self.create_augmentation_functions()
augmented_samples = []
aug_labels = []

for aug_name, aug_fn in list(augmentation_fns.items())[:4]:
    if aug_name in ['mixup', 'cutmix']:
        aug_sample = aug_fn(original, X_test[1], 0.5)
    else:
        aug_sample = aug_fn(original, 0.3)
    augmented_samples.append(aug_sample[:20]) # First 20 features
    aug_labels.append(aug_name)

# Plot
x = np.arange(20)
ax.plot(x, original[:20], 'k-', linewidth=2, label='Original')

for aug_sample, aug_label in zip(augmented_samples, aug_labels):
    ax.plot(x, aug_sample, '--', alpha=0.7, label=aug_label)

ax.set_xlabel('Feature Index')
ax.set_ylabel('Value')
ax.set_title('Augmentation Examples')
ax.legend()
ax.grid(True, alpha=0.3)

# 5. Strategy correlation
ax = axes[1, 1]

if len(self.augmentation_history) > 10:
    # Extract strategy matrix
    aug_types = list(self.augmentation_space.keys())
    strategy_matrix = np.zeros((len(self.augmentation_history),
                                len(aug_types)))

```

484

```

        for i, strategy in enumerate(self.augmentation_history):
            for j, aug_type in enumerate(aug_types):
                if aug_type in strategy:
                    strategy_matrix[i, j] = strategy[aug_type]

    # Correlate with performance
    correlations = []
    for j in range(len(aug_types)):
        corr = np.corrcoef(strategy_matrix[:, j],
                            self.performance_history)[0, 1]
        correlations.append(corr)

    bars = ax.bar(aug_types, correlations)

    # Color by correlation
    for bar, corr in zip(bars, correlations):
        if corr > 0:
            bar.set_color('green')
        else:
            bar.set_color('red')

    ax.set_xlabel('Augmentation Type')
    ax.set_ylabel('Correlation with Performance')
    ax.set_title('Augmentation Impact Analysis')
    ax.tick_params(axis='x', rotation=45)
    ax.grid(True, alpha=0.3, axis='y')

# 6. Adversarial robustness
ax = axes[1, 2]

# Create simple model for demonstration
model = keras.Sequential([
    layers.Dense(64, activation='relu', input_shape=
(X_test.shape[1],)),
    layers.Dense(32, activation='relu'),
    layers.Dense(len(np.unique(y_test)), activation='softmax')
])
model.compile(optimizer='adam',
loss='sparse_categorical_crossentropy')

# Test adversarial robustness
epsilons = [0, 0.05, 0.1, 0.15, 0.2]
robustness_scores = []

for eps in epsilons:
    if eps == 0:
        X_test_adv = X_test[:100]
    else:
        X_test_adv = self.apply_adversarial_perturbation(
            X_test[:100], model, epsilon=eps
)

```

```

        )

    # Measure perturbation effect
    perturbation_norm = np.mean(np.linalg.norm(X_test_adv -
X_test[:100], axis=1))
    robustness_scores.append(perturbation_norm)

    ax.plot(epsilons, robustness_scores, 'r-', linewidth=2,
marker='o')
    ax.set_xlabel('Adversarial Epsilon')
    ax.set_ylabel('Average Perturbation Norm')
    ax.set_title('Adversarial Robustness Testing')
    ax.grid(True, alpha=0.3)

    plt.suptitle('Advanced Data Augmentation Analysis', fontsize=16)
    plt.tight_layout()
    plt.show()

# Test adaptive data augmentation
def test_adaptive_augmentation():
    # Generate synthetic data
    from sklearn.datasets import make_classification

    X, y = make_classification(
        n_samples=1000,
        n_features=20,
        n_informative=15,
        n_redundant=5,
        n_classes=3,
        random_state=42
    )

    # Split data
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=42
    )
    X_train, X_val, y_train, y_val = train_test_split(
        X_train, y_train, test_size=0.2, random_state=42
    )

    # Define augmentation space
    augmentation_space = {
        'noise': {'min': 0.0, 'max': 0.5},
        'scaling': {'min': 0.0, 'max': 0.3},
        'dropout': {'min': 0.0, 'max': 0.3},
        'mixup': {'min': 0.0, 'max': 0.8},
        'adversarial': {'min': 0.0, 'max': 0.2}
    }

    # Create augmentation system

```

486

```

augmenter = AdaptiveDataAugmentation(augmentation_space)

# Define base model function
def base_model_fn():
    model = keras.Sequential([
        layers.Dense(64, activation='relu', input_shape=(20,)),
        layers.Dropout(0.3),
        layers.Dense(32, activation='relu'),
        layers.Dropout(0.3),
        layers.Dense(3, activation='softmax')
    ])

    model.compile(
        optimizer='adam',
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy']
    )

    return model

# Learn augmentation policy
print("Learning optimal augmentation policy...")
learned_policy = augmenter.learn_augmentation_policy(
    X_train, y_train, X_val, y_val,
    base_model_fn, n_trials=20
) 487

print("\nLearned Policy:")
for aug_name, policy in learned_policy.items():
    print(f" {aug_name}: Use prob={policy['use_probability']:.2f}, "
          f"Strength={policy['strength']:.3f}")

# Generate synthetic data
X_synthetic, y_synthetic = augmenter.generate_synthetic_data(
    X_train, y_train, n_synthetic=200
)

print(f"\nGenerated {len(X_synthetic)} synthetic samples")

# Test adaptive training
model = base_model_fn()
adapted_history = augmenter.adapt_during_training(
    model, X_train, y_train, X_val, y_val
)

# Visualize analysis
augmenter.visualize_augmentation_analysis(X_test, y_test)

if __name__ == "__main__":
    test_adaptive_augmentation()

```



**Task:** Implement an AutoML system that searches architectures and preprocessing pipelines.

```
import itertools
from sklearn.preprocessing import StandardScaler, MinMaxScaler,
RobustScaler
from sklearn.model_selection import train_test_split
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import matplotlib.pyplot as plt
import time
import random

class SecurityAutoML:
    """
        AutoML system for security applications with NAS and pipeline
    optimization
    """
    def __init__(self, task_type='classification'):
        self.task_type = task_type
        self.search_results = []
        self.best_model = None
        self.best_pipeline = None
        self.pareto_frontier = []

    def define_search_spaces(self):
        """Define search spaces for architecture and preprocessing"""

        # Architecture search space
        architecture_space = {
            'n_layers': [2, 3, 4, 5, 6],
            'layer_sizes': [32, 64, 128, 256, 512],
            'activation': ['relu', 'elu', 'tanh', 'swish'],
            'optimizer': ['adam', 'rmsprop', 'sgd'],
            'learning_rate': [0.0001, 0.001, 0.01],
            'batch_norm': [True, False],
            'dropout_rate': [0.0, 0.2, 0.3, 0.5],
            'regularization': ['none', 'l1', 'l2', 'l1_l2']
        }

        # Preprocessing search space
        preprocessing_space = {
            'scaler': [StandardScaler(), MinMaxScaler(), RobustScaler(),
None],
            'feature_selection': ['none', 'variance', 'correlation',
'mutual_info'],
        }
```

```

        'feature_threshold': [0.01, 0.05, 0.1],
        'pca': [None, 0.95, 0.99], # None or variance explained
    }

    # Loss function search space
    if self.task_type == 'classification':
        loss_space = {
            'loss': ['categorical_crossentropy', 'focal_loss',
'weighted_ce'],
            'class_weight': ['none', 'balanced', 'custom']
        }
    else:
        loss_space = {
            'loss': ['mse', 'mae', 'huber', 'log_cosh'],
            'metric': ['mse', 'mae', 'mape']
        }

    return architecture_space, preprocessing_space, loss_space

def create_model_from_config(self, arch_config, input_shape,
output_shape):
    """Create model from architecture configuration"""

    model = keras.Sequential()
    model.add(layers.Input(shape=input_shape))

    # Build layers
    layer_sizes = []
    for i in range(arch_config['n_layers']):
        # Gradually decrease layer size
        size = int(arch_config['layer_sizes'] // (i + 1))
        layer_sizes.append(size)

        # Add dense layer
        if arch_config['regularization'] == 'l1':
            model.add(layers.Dense(size, activation=None,
kernel_regularizer=keras.regularizers.l1(0.01)))
        elif arch_config['regularization'] == 'l2':
            model.add(layers.Dense(size, activation=None,
kernel_regularizer=keras.regularizers.l2(0.01)))
        elif arch_config['regularization'] == 'l1_l2':
            model.add(layers.Dense(size, activation=None,
kernel_regularizer=keras.regularizers.l1_l2(0.01, 0.01)))
        else:
            model.add(layers.Dense(size, activation=None))

    # Batch normalization

```

```

        if arch_config['batch_norm']:
            model.add(layers.BatchNormalization())

        # Activation
        if arch_config['activation'] == 'swish':
            model.add(layers.Activation(tf.nn.swish))
        else:
            model.add(layers.Activation(arch_config['activation']))

        # Dropout
        if arch_config['dropout_rate'] > 0:
            model.add(layers.Dropout(arch_config['dropout_rate']))

    # Output layer
    if self.task_type == 'classification':
        model.add(layers.Dense(output_shape, activation='softmax'))
    else:
        model.add(layers.Dense(output_shape, activation='linear'))

    return model, layer_sizes

def create_preprocessing_pipeline(self, config, X):
    """Create preprocessing pipeline from configuration"""

    from sklearn.pipeline import Pipeline
    from sklearn.feature_selection import VarianceThreshold,
    SelectKBest
    from sklearn.feature_selection import mutual_info_classif
    from sklearn.decomposition import PCA

    steps = []

    # Scaling
    if config['scaler'] is not None:
        steps.append(('scaler', config['scaler']))

    # Feature selection
    if config['feature_selection'] == 'variance':
        steps.append(('variance_selection',
                      VarianceThreshold(threshold=config['feature_threshold'])))

    elif config['feature_selection'] == 'mutual_info':
        steps.append(('mutual_info_selection',
                      SelectKBest(mutual_info_classif, k='all')))

    # PCA
    if config['pca'] is not None:
        steps.append(('pca', PCA(n_components=config['pca'])))

    if steps:

```

```

        pipeline = Pipeline(steps)
    else:
        pipeline = None

    return pipeline

    def implement_neural_architecture_search(self, X_train, y_train,
X_val, y_val,
   n_trials=50,
search_strategy='random'):
        """Implement NAS with different search strategies"""

        arch_space, prep_space, loss_space = self.define_search_spaces()

        if search_strategy == 'random':
            configs = self._random_search(arch_space, prep_space,
loss_space, n_trials)
        elif search_strategy == 'evolutionary':
            configs = self._evolutionary_search(arch_space, prep_space,
loss_space, n_trials)
        elif search_strategy == 'bayesian':
            configs = self._bayesian_search(arch_space, prep_space,
loss_space, n_trials)
        else:
            raise ValueError(f"Unknown search strategy:
{search_strategy}")

        # Evaluate configurations
        for i, (arch_config, prep_config, loss_config) in
enumerate(configs):
            print(f"\nEvaluating configuration {i+1}/{len(configs)}...")

        # Create preprocessing pipeline
        pipeline = self.create_preprocessing_pipeline(prep_config,
X_train)

        # Transform data
        if pipeline:
            X_train_transformed = pipeline.fit_transform(X_train)
            X_val_transformed = pipeline.transform(X_val)
        else:
            X_train_transformed = X_train
            X_val_transformed = X_val

        # Create model
        input_shape = (X_train_transformed.shape[1],)
        output_shape = len(np.unique(y_train)) if self.task_type ==
'classification' else 1

        model, layer_sizes = self.create_model_from_config(

```

```

        arch_config, input_shape, output_shape
    )

    # Compile model
    self._compile_model(model, arch_config, loss_config)

    # Train model
    start_time = time.time()
    history = model.fit(
        X_train_transformed, y_train,
        validation_data=(X_val_transformed, y_val),
        epochs=20,
        batch_size=32,
        verbose=0
    )
    train_time = time.time() - start_time

    # Get metrics
    if self.task_type == 'classification':
        val_loss, val_acc = model.evaluate(X_val_transformed,
y_val, verbose=0)
        metric = val_acc
    else:
        val_loss, val_metric = model.evaluate(X_val_transformed,
y_val, verbose=0)
        metric = -val_loss # Minimize loss
    493

    # Calculate model complexity
    total_params = model.count_params()

    # Store results
    result = {
        'arch_config': arch_config,
        'prep_config': prep_config,
        'loss_config': loss_config,
        'metric': metric,
        'train_time': train_time,
        'total_params': total_params,
        'layer_sizes': layer_sizes,
        'history': history
    }

    self.search_results.append(result)

    print(f"Metric: {metric:.4f}, Params: {total_params:}, Time:
{train_time:.2f}s")

    # Find best model and Pareto frontier
    self._find_best_and_pareto()

```

```

    def _random_search(self, arch_space, prep_space, loss_space,
n_trials):
        """Random search strategy"""
        configs = []

        for _ in range(n_trials):
            arch_config = {k: random.choice(v) for k, v in
arch_space.items()}
            prep_config = {k: random.choice(v) if k != 'scaler' else
v[random.randint(0, len(v)-1)]
                           for k, v in prep_space.items()}
            loss_config = {k: random.choice(v) for k, v in
loss_space.items()}

            configs.append((arch_config, prep_config, loss_config))

        return configs

    def _evolutionary_search(self, arch_space, prep_space, loss_space,
n_trials):
        """Evolutionary search strategy"""
        # Simplified evolutionary algorithm
        population_size = 20
        n_generations = n_trials // population_size

        # Initialize population
        population = []
        for _ in range(population_size):
            arch_config = {k: random.choice(v) for k, v in
arch_space.items()}
            prep_config = {k: random.choice(v) if k != 'scaler' else
v[random.randint(0, len(v)-1)]
                           for k, v in prep_space.items()}
            loss_config = {k: random.choice(v) for k, v in
loss_space.items()}

            population.append((arch_config, prep_config, loss_config))

        all_configs = population.copy()

        # Evolution
        for gen in range(n_generations - 1):
            # Create offspring through mutation
            offspring = []
            for parent in population[:population_size//2]: # Top half
                child = self._mutate_config(parent, arch_space,
prep_space, loss_space)
                offspring.append(child)

            population.extend(offspring)
            all_configs.extend(offspring)

```

```

    return all_configs

def _mutate_config(self, config, arch_space, prep_space, loss_space):
    """Mutate configuration"""
    arch_config, prep_config, loss_config = config

    # Mutate one parameter
    if random.random() < 0.5:
        # Mutate architecture
        param = random.choice(list(arch_config.keys()))
        arch_config = arch_config.copy()
        arch_config[param] = random.choice(arch_space[param])
    else:
        # Mutate preprocessing
        param = random.choice(list(prep_config.keys()))
        prep_config = prep_config.copy()
        if param == 'scaler':
            prep_config[param] = prep_space[param][random.randint(0,
len(prep_space[param])-1)]
        else:
            prep_config[param] = random.choice(prep_space[param])

    return (arch_config, prep_config, loss_config)

```

495

```

def _bayesian_search(self, arch_space, prep_space, loss_space,
n_trials):
    """Bayesian optimization search (simplified)"""
    # For simplicity, using random with preference for good regions
    configs = self._random_search(arch_space, prep_space, loss_space,
n_trials)
    return configs

def _compile_model(self, model, arch_config, loss_config):
    """Compile model with specified configuration"""
    # Get optimizer
    if arch_config['optimizer'] == 'adam':
        optimizer =
keras.optimizers.Adam(learning_rate=arch_config['learning_rate'])
    elif arch_config['optimizer'] == 'rmsprop':
        optimizer =
keras.optimizers.RMSprop(learning_rate=arch_config['learning_rate'])
    else:
        optimizer =
keras.optimizers.SGD(learning_rate=arch_config['learning_rate'])

    # Get loss
    if loss_config['loss'] == 'focal_loss':
        # Custom focal loss
        def focal_loss(y_true, y_pred, gamma=2.0, alpha=0.25):

```

```

        epsilon = keras.backend.epsilon()
        y_pred = keras.backend.clip(y_pred, epsilon, 1. - epsilon)

        # Calculate focal loss
        p_t = tf.where(keras.backend.equal(y_true, 1), y_pred, 1 - y_pred)
        alpha_factor = keras.backend.ones_like(y_true) * alpha
        alpha_t = tf.where(keras.backend.equal(y_true, 1),
                           alpha_factor, 1 - alpha_factor)
        cross_entropy = -keras.backend.log(p_t)
        weight = alpha_t * keras.backend.pow((1 - p_t), gamma)
        loss = weight * cross_entropy
        return keras.backend.mean(keras.backend.sum(loss, axis=1))

    loss = focal_loss
else:
    loss = loss_config['loss']

# Compile
if self.task_type == 'classification':
    metrics = ['accuracy']
else:
    metrics = ['mae']

model.compile(optimizer=optimizer, loss=loss, metrics=metrics) 496

def _find_best_and_pareto(self):
    """Find best model and Pareto frontier"""
    # Best by metric
    self.best_model = max(self.search_results, key=lambda x:
x['metric'])

    # Pareto frontier (metric vs complexity)
    self.pareto_frontier = []

    for result in self.search_results:
        is_pareto = True
        for other in self.search_results:
            if (other['metric'] > result['metric'] and
                other['total_params'] < result['total_params']):
                is_pareto = False
                break

        if is_pareto:
            self.pareto_frontier.append(result)

def visualize_search_results(self):
    """Visualize AutoML search results"""

fig, axes = plt.subplots(2, 3, figsize=(15, 10))

```

```

# 1. Metric vs Parameters scatter
ax = axes[0, 0]

metrics = [r['metric'] for r in self.search_results]
params = [r['total_params'] for r in self.search_results]

ax.scatter(params, metrics, alpha=0.6, label='All models')

# Highlight Pareto frontier
if self.pareto_frontier:
    pareto_params = [r['total_params'] for r in
self.pareto_frontier]
    pareto_metrics = [r['metric'] for r in self.pareto_frontier]
    ax.scatter(pareto_params, pareto_metrics, color='red', s=100,
               label='Pareto optimal', zorder=10)

# Connect Pareto points
sorted_indices = np.argsort(pareto_params)
ax.plot([pareto_params[i] for i in sorted_indices],
         [pareto_metrics[i] for i in sorted_indices],
         'r--', alpha=0.5)

# Highlight best
ax.scatter(self.best_model['total_params'],
           self.best_model['metric'],
           color='gold', s=200, marker='*', label='Best model',
           zorder=20)

ax.set_xlabel('Total Parameters')
ax.set_ylabel('Validation Metric')
ax.set_title('Model Performance vs Complexity')
ax.legend()
ax.grid(True, alpha=0.3)

# 2. Training time analysis
ax = axes[0, 1]

train_times = [r['train_time'] for r in self.search_results]

ax.scatter(params, train_times, alpha=0.6, c=metrics,
cmap='viridis')
ax.set_xlabel('Total Parameters')
ax.set_ylabel('Training Time (s)')
ax.set_title('Training Time vs Model Size')
im = ax.scatter(params, train_times, alpha=0.6, c=metrics,
cmap='viridis')
plt.colorbar(im, ax=ax, label='Metric')
ax.grid(True, alpha=0.3)

```

```

# 3. Architecture distribution
ax = axes[0, 2]

n_layers_dist = [r['arch_config']['n_layers'] for r in
self.search_results]
unique_layers, counts = np.unique(n_layers_dist,
return_counts=True)

ax.bar(unique_layers, counts, color='skyblue', alpha=0.8)
ax.set_xlabel('Number of Layers')
ax.set_ylabel('Count')
ax.set_title('Architecture Depth Distribution')
ax.grid(True, alpha=0.3, axis='y')

# 4. Hyperparameter importance
ax = axes[1, 0]

# Calculate correlation with metric for each hyperparameter
hp_importance = {}

# Numerical hyperparameters
for hp in ['n_layers', 'layer_sizes', 'learning_rate',
'dropout_rate']:
    values = [r['arch_config'][hp] for r in self.search_results]
    if isinstance(values[0], (int, float)):
        correlation = np.corrcoef(values, metrics)[0, 1]
        hp_importance[hp] = abs(correlation) 498

hp_names = list(hp_importance.keys())
hp_values = list(hp_importance.values())

bars = ax.bar(hp_names, hp_values, color='green', alpha=0.8)
ax.set_xlabel('Hyperparameter')
ax.set_ylabel('Absolute Correlation with Metric')
ax.set_title('Hyperparameter Importance')
ax.tick_params(axis='x', rotation=45)
ax.grid(True, alpha=0.3, axis='y')

# 5. Loss function comparison
ax = axes[1, 1]

loss_types = [r['loss_config']['loss'] for r in
self.search_results]
unique_losses = list(set(loss_types))

loss_metrics = {}
for loss_type in unique_losses:
    loss_metrics[loss_type] = [
        r['metric'] for r in self.search_results
        if r['loss_config']['loss'] == loss_type

```

```

        ]

    ax.boxplot(loss_metrics.values(), labels=loss_metrics.keys())
    ax.set_xlabel('Loss Function')
    ax.set_ylabel('Validation Metric')
    ax.set_title('Loss Function Performance')
    ax.tick_params(axis='x', rotation=45)
    ax.grid(True, alpha=0.3, axis='y')

# 6. Best model details
ax = axes[1, 2]

best_config_text = "Best Model Configuration\n\n"
best_config_text += "Architecture:\n"
for k, v in self.best_model['arch_config'].items():
    best_config_text += f" {k}: {v}\n"

best_config_text += "\nPreprocessing:\n"
for k, v in self.best_model['prep_config'].items():
    if k != 'scaler':
        best_config_text += f" {k}: {v}\n"
    else:
        best_config_text += f" {k}: {type(v).__name__} if v else 'None'\n"

best_config_text += f"\nMetric: {self.best_model['metric']:.4f}"
best_config_text += f"\nParameters:\n{self.best_model['total_params']}"

ax.text(0.1, 0.9, best_config_text, transform=ax.transAxes,
        fontsize=9, verticalalignment='top',
        fontfamily='monospace')
ax.axis('off')

plt.suptitle('AutoML Search Results', fontsize=16)
plt.tight_layout()
plt.show()

# Test AutoML system
def test_automl_security():
    # Generate synthetic security dataset
    from sklearn.datasets import make_classification

    X, y = make_classification(
        n_samples=2000,
        n_features=50,
        n_informative=30,
        n_redundant=10,
        n_classes=5,
        flip_y=0.1,

```

```
    random_state=42
)

# Split data
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)
X_train, X_val, y_train, y_val = train_test_split(
    X_train, y_train, test_size=0.2, random_state=42
)

# Create AutoML system
automl = SecurityAutoML(task_type='classification')

# Run search
print("Starting AutoML search...")
automl.implement_neural_architecture_search(
    X_train, y_train, X_val, y_val,
    n_trials=30,
    search_strategy='evolutionary'
)

# Visualize results
automl.visualize_search_results()

# Print Pareto frontier
print(f"\nFound {len(automl.pareto_frontier)} Pareto optimal models")
print("\nTop 5 models by metric:")
sorted_results = sorted(automl.search_results, key=lambda x:
x['metric'], reverse=True)[:5]

for i, result in enumerate(sorted_results):
    print(f"\n{i+1}. Metric: {result['metric']:.4f}, "
          f"Params: {result['total_params']:,}, "
          f"Layers: {result['arch_config']['n_layers']}")

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

**Task:** Build a complete API with versioning, A/B testing, and monitoring.

```
from flask import Flask, request, jsonify
import redis
import pickle
import uuid
from datetime import datetime
import prometheus_client
from prometheus_client import Counter, Histogram, Gauge
import logging
import json
import numpy as np
import matplotlib.pyplot as plt
import psutil

class ProductionMLAPI:
    """
    Production-ready ML API with monitoring and A/B testing
    """

    def __init__(self, redis_host='localhost', redis_port=6379):
        self.app = Flask(__name__)
        try:
            self.redis_client = redis.Redis(host=redis_host,
port=redis_port)
            self.redis_client.ping()
        except:
            print("Warning: Redis not available, using in-memory storage")
            self.redis_client = None

        self.models = {}
        self.model_versions = {}
        self.ab_tests = {}

        # In-memory storage if Redis not available
        self.memory_storage = []

        # Metrics
        self.prediction_counter = Counter('ml_predictions_total',
   'Total predictions',
   ['model_version', 'endpoint'])
        self.prediction_latency =
Histogram('ml_prediction_latency_seconds',
           'Prediction latency',
           ['model_version'])
        self.active_models = Gauge('ml_active_models',
           'Number of active models')
        self.error_counter = Counter('ml_errors_total',
           'Total errors',
```

```
[ 'error_type' ])
```

```
# Logging
self.setup_logging()
```

```
# Routes
self.setup_routes()
```

```
def setup_logging(self):
    """Setup comprehensive logging"""
    logging.basicConfig(
        level=logging.INFO,
        format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
        handlers=[
            logging.FileHandler('ml_api.log'),
            logging.StreamHandler()
        ]
    )
    self.logger = logging.getLogger(__name__)
```

```
def setup_routes(self):
    """Setup API routes"""

    @self.app.route('/health', methods=['GET'])
    def health_check():
        """Health check endpoint"""
        redis_status = False
        if self.redis_client:
            try:
                redis_status = self.redis_client.ping()
            except:
                redis_status = False

        return jsonify({
            'status': 'healthy',
            'timestamp': datetime.now().isoformat(),
            'active_models': len(self.models),
            'redis_connected': redis_status
        })

    @self.app.route('/predict', methods=['POST'])
    def predict():
        """Main prediction endpoint"""
        try:
            # Get request data
            data = request.get_json()

            # Validate input
            validation_result = self.validate_input(data)
            if not validation_result['valid']:
```

```

        return jsonify({
            'error': validation_result['message']
        }), 400

    # Get model version (A/B testing)
    model_version =
self.get_model_version(data.get('user_id'))

    # Make prediction
    import time
    start_time = time.time()
    prediction = self.make_prediction(
        data['features'],
        model_version
    )
    latency = time.time() - start_time

    # Update metrics
    self.prediction_counter.labels(
        model_version=model_version,
        endpoint='predict'
    ).inc()
    self.prediction_latency.labels(
        model_version=model_version
    ).observe(latency)

    # Log prediction
    self.log_prediction(data, prediction, model_version,
latency)

    # Return response
    return jsonify({
        'prediction': prediction,
        'model_version': model_version,
        'latency_ms': latency * 1000,
        'request_id': str(uuid.uuid4())
    })

except Exception as e:

self.error_counter.labels(error_type=type(e).__name__).inc()
    self.logger.error(f"Prediction error: {str(e)}")
    return jsonify({
        'error': 'Internal server error',
        'request_id': str(uuid.uuid4())
    }), 500

@self.app.route('/models', methods=['GET'])
def list_models():
    """List all model versions"""

```

```

        return jsonify({
            'models': list(self.model_versions.keys()),
            'active_models': len(self.models),
            'ab_tests': list(self.ab_tests.keys())
        })

    @self.app.route('/models/<version>', methods=['POST'])
    def deploy_model(version):
        """Deploy a new model version"""
        try:
            # For demo, create a dummy model
            from sklearn.ensemble import RandomForestClassifier
            model = RandomForestClassifier(n_estimators=10)

            # Store model
            self.models[version] = model
            self.model_versions[version] = {
                'deployed_at': datetime.now().isoformat(),
                'status': 'active'
            }

            # Update metrics
            self.active_models.set(len(self.models))

            self.logger.info(f"Deployed model version: {version}"))

            return jsonify({
                'message': f'Model {version} deployed successfully',
                'version': version
            })
        except Exception as e:
            self.logger.error(f"Deployment error: {str(e)}")
            return jsonify({
                'error': f'Failed to deploy model: {str(e)}'
            }), 500

    @self.app.route('/ab_test', methods=['POST'])
    def create_ab_test():
        """Create A/B test"""
        data = request.get_json()

        test_id = str(uuid.uuid4())
        self.ab_tests[test_id] = {
            'model_a': data['model_a'],
            'model_b': data['model_b'],
            'traffic_split': data.get('traffic_split', 0.5),
            'created_at': datetime.now().isoformat()
        }

```

```

        return jsonify({
            'test_id': test_id,
            'message': 'A/B test created successfully'
        })

    @self.app.route('/metrics', methods=['GET'])
    def metrics():
        """Prometheus metrics endpoint"""
        return prometheus_client.generate_latest()

    @self.app.route('/monitoring/dashboard', methods=['GET'])
    def monitoring_dashboard():
        """Real-time monitoring dashboard data"""
        # Get recent predictions
        recent_predictions = self.get_recent_predictions()

        # Calculate metrics
        metrics = {
            'total_predictions': len(recent_predictions),
            'avg_latency': np.mean([p['latency'] for p in
recent_predictions]) if recent_predictions else 0,
            'error_rate': sum(1 for p in recent_predictions if
p.get('error')) / max(len(recent_predictions), 1),
            'predictions_per_model': {}
        }

        # Count by model version
        for pred in recent_predictions:
            version = pred.get('model_version', 'unknown')
            metrics['predictions_per_model'][version] =
metrics['predictions_per_model'].get(version, 0) + 1

        return jsonify(metrics)

    def validate_input(self, data):
        """Validate input data"""
        if not data:
            return {'valid': False, 'message': 'No data provided'}

        if 'features' not in data:
            return {'valid': False, 'message': 'No features provided'}

        features = data['features']

        # Check feature type and shape
        if not isinstance(features, list):
            return {'valid': False, 'message': 'Features must be a list'}

        # Check for required feature count
        if len(features) == 0:

```

```
        return {'valid': False, 'message': 'Empty feature list'}
```

```
    # Check for NaN or infinite values
    import math
    for i, val in enumerate(features):
        if not isinstance(val, (int, float)):
            return {'valid': False, 'message': f'Feature {i} is not numeric'}
        if math.isnan(val) or math.isinf(val):
            return {'valid': False, 'message': f'Feature {i} is NaN or infinite'}
```

```
    return {'valid': True}
```

```
def get_model_version(self, user_id=None):
    """Get model version for user (A/B testing)"""
    # Check if user is in an A/B test
    if user_id and self.ab_tests:
        # Simple hash-based assignment
        user_hash = hash(user_id) % 100

        for test_id, test_config in self.ab_tests.items():
            traffic_split = test_config['traffic_split'] * 100

            if user_hash < traffic_split:
                return test_config['model_a']
            else:
                return test_config['model_b']

    # Return latest model version
    if self.model_versions:
        return max(self.model_versions.keys())

    return 'default'
```

```
def make_prediction(self, features, model_version):
    """Make prediction using specified model version"""
    if model_version not in self.models:
        # For demo, return random prediction
        return float(np.random.random())

    model = self.models[model_version]

    # Convert features to numpy array
    features_array = np.array(features).reshape(1, -1)

    # For demo, return random prediction
    prediction = np.random.random()

    return float(prediction)
```

```
def log_prediction(self, request_data, prediction, model_version,
latency):
    """Log prediction for analysis"""
    log_entry = {
        'timestamp': datetime.now().isoformat(),
        'request_id': str(uuid.uuid4()),
        'model_version': model_version,
        'features': request_data['features'],
        'prediction': prediction,
        'latency': latency,
        'user_id': request_data.get('user_id')
    }

    # Store in Redis or memory
    if self.redis_client:
        try:
            self.redis_client.lpush('predictions',
pickle.dumps(log_entry))
            self.redis_client.ltrim('predictions', 0, 10000) # Keep
last 10k
        except:
            self.memory_storage.append(log_entry)
    else:
        self.memory_storage.append(log_entry)
        # Keep last 10k in memory
        if len(self.memory_storage) > 10000:
            self.memory_storage = self.memory_storage[-10000:]

    # Log to file
    self.logger.info(f"Prediction: {log_entry}")

def get_recent_predictions(self, n=100):
    """Get recent predictions from storage"""
    predictions = []

    if self.redis_client:
        try:
            for i in range(min(n,
self.redis_client.llen('predictions'))):
                pred = self.redis_client.lindex('predictions', i)
                if pred:
                    predictions.append(pickle.loads(pred))
        except:
            # Fallback to memory
            predictions = self.memory_storage[-n:]
    else:
        predictions = self.memory_storage[-n:]

    return predictions
```

```
def auto_scale(self, cpu_threshold=80, memory_threshold=90):
    """Auto-scaling logic"""
    # Get current resource usage
    cpu_percent = psutil.cpu_percent(interval=1)
    memory_percent = psutil.virtual_memory().percent

    scaling_decision = {
        'scale_up': False,
        'scale_down': False,
        'current_cpu': cpu_percent,
        'current_memory': memory_percent
    }

    if cpu_percent > cpu_threshold or memory_percent > memory_threshold:
        scaling_decision['scale_up'] = True
        self.logger.warning(f"High resource usage - CPU: {cpu_percent}%, "
                            f"Memory: {memory_percent}%")
    elif cpu_percent < 20 and memory_percent < 30:
        scaling_decision['scale_down'] = True

    return scaling_decision

def run(self, host='0.0.0.0', port=5000):
    """Run the API server"""
    self.app.run(host=host, port=port)

# Create comprehensive deployment script
def create_deployment_configurations():
    """Create Docker and Kubernetes deployment configurations"""

    # Dockerfile
    dockerfile_content = """
FROM python:3.8-slim

WORKDIR /app

# Install dependencies
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copy application
COPY . .

# Expose port
EXPOSE 5000

# Health check
    """

    # Create Dockerfile
    with open('Dockerfile', 'w') as file:
        file.write(dockerfile_content)

    # Create deployment configuration files
    k8s_namespace = os.getenv('K8S_NAMESPACE', 'default')
    k8s_name = os.getenv('K8S_NAME', 'api-server')

    # Create deployment manifest
    deployment_manifest = f"""
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {k8s_name}
  namespace: {k8s_namespace}
spec:
  selector:
    matchLabels:
      app: {k8s_name}
  replicas: 1
  template:
    metadata:
      labels:
        app: {k8s_name}
    spec:
      containers:
        - name: {k8s_name}
          image: {host}:{port}
          ports:
            - containerPort: 5000
      imagePullSecrets:
        - name: registry-secret
    """
    with open(f'{k8s_name}-deployment.yaml', 'w') as file:
        file.write(deployment_manifest)

    # Create service manifest
    service_manifest = f"""
apiVersion: v1
kind: Service
metadata:
  name: {k8s_name}-service
  namespace: {k8s_namespace}
spec:
  selector:
    app: {k8s_name}
  ports:
    - name: http
      port: 5000
      targetPort: 5000
  type: ClusterIP
    """
    with open(f'{k8s_name}-service.yaml', 'w') as file:
        file.write(service_manifest)
```

508

```
HEALTHCHECK --interval=30s --timeout=3s --start-period=5s --retries=3 \
  CMD curl -f http://localhost:5000/health || exit 1

# Run application
CMD ["python", "api.py"]
"""

# Kubernetes deployment
k8s_deployment = """
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ml-api
  labels:
    app: ml-api
spec:
  replicas: 3
  selector:
    matchLabels:
      app: ml-api
  template:
    metadata:
      labels:
        app: ml-api
  spec: 509
    containers:
      - name: ml-api
        image: ml-api:latest
        ports:
          - containerPort: 5000
        resources:
          requests:
            memory: "512Mi"
            cpu: "500m"
          limits:
            memory: "1Gi"
            cpu: "1000m"
        livenessProbe:
          httpGet:
            path: /health
            port: 5000
          initialDelaySeconds: 30
          periodSeconds: 10
        readinessProbe:
          httpGet:
            path: /health
            port: 5000
          initialDelaySeconds: 5
          periodSeconds: 5
...
"""
```

```
apiVersion: v1
kind: Service
metadata:
  name: ml-api-service
spec:
  selector:
    app: ml-api
  ports:
    - protocol: TCP
      port: 80
      targetPort: 5000
  type: LoadBalancer
---
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: ml-api-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: ml-api
  minReplicas: 2
  maxReplicas: 10
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 70
    - type: Resource
      resource:
        name: memory
        target:
          type: Utilization
          averageUtilization: 80
    ...
# Docker compose for development
docker_compose = """
version: '3.8'

services:
  ml-api:
    build: .
    ports:
      - "5000:5000"
    environment:
      - REDIS_HOST=redis
      - REDIS_PASSWORD=redis
      - REDIS_PORT=6379
```

```

    - PROMETHEUS_MULTIPROC_DIR=/tmp
depends_on:
  - redis
volumes:
  - ./models:/app/models
  - ./logs:/app/logs

redis:
  image: redis:alpine
  ports:
    - "6379:6379"

prometheus:
  image: prom/prometheus
  ports:
    - "9090:9090"
  volumes:
    - ./prometheus.yml:/etc/prometheus/prometheus.yml

grafana:
  image: grafana/grafana
  ports:
    - "3000:3000"
  environment:
    - GF_SECURITY_ADMIN_PASSWORD=admin
"""

# Requirements.txt
requirements = """
flask==2.0.1
redis==3.5.3
prometheus-client==0.11.0
numpy==1.21.0
scikit-learn==0.24.2
tensorflow==2.6.0
psutil==5.8.0
matplotlib==3.4.2
"""

"""

print("== Deployment Configurations Created ==")
print("\n1. Dockerfile - for containerization")
print("2. Kubernetes manifests - for orchestration")
print("3. Docker Compose - for local development")
print("4. Requirements.txt - Python dependencies")
print("\nFeatures included:")
print("- Auto-scaling based on CPU/memory")
print("- Health checks and readiness probes")
print("- Resource limits and requests")
print("- Load balancing")
print("- Monitoring with Prometheus/Grafana")

```

```
return {
    'dockerfile': dockerfile_content,
    'k8s_deployment': k8s_deployment,
    'docker_compose': docker_compose,
    'requirements': requirements
}

# Visualize production monitoring
def visualize_production_monitoring():
    """Create production monitoring dashboard visualization"""

    fig, axes = plt.subplots(2, 3, figsize=(15, 10))

    # Simulate production metrics
    time_points = np.arange(0, 100)

    # 1. Request rate
    ax = axes[0, 0]
    request_rate = 100 + 50 * np.sin(time_points / 10) +
    np.random.normal(0, 10, 100)
    ax.plot(time_points, request_rate, 'b-', linewidth=2)
    ax.fill_between(time_points, request_rate, alpha=0.3)
    ax.set_xlabel('Time (minutes)')
    ax.set_ylabel('Requests/second')
    ax.set_title('Request Rate')
    ax.grid(True, alpha=0.3)

    # 2. Latency percentiles
    ax = axes[0, 1]
    p50 = 20 + np.random.normal(0, 2, 100)
    p95 = 50 + np.random.normal(0, 5, 100)
    p99 = 100 + np.random.normal(0, 10, 100)

    ax.plot(time_points, p50, 'g-', label='P50', linewidth=2)
    ax.plot(time_points, p95, 'y-', label='P95', linewidth=2)
    ax.plot(time_points, p99, 'r-', label='P99', linewidth=2)
    ax.set_xlabel('Time (minutes)')
    ax.set_ylabel('Latency (ms)')
    ax.set_title('Response Latency')
    ax.legend()
    ax.grid(True, alpha=0.3)

    # 3. Error rate
    ax = axes[0, 2]
    error_rate = np.random.exponential(0.01, 100) * 100
    ax.plot(time_points, error_rate, 'r-', linewidth=2)
    ax.axhline(y=1, color='orange', linestyle='--', label='Warning
threshold')
    ax.axhline(y=5, color='red', linestyle='--', label='Critical
```

```
threshold')

ax.set_xlabel('Time (minutes)')
ax.set_ylabel('Error Rate (%)')
ax.set_title('Error Rate')
ax.legend()
ax.grid(True, alpha=0.3)

# 4. Model version distribution
ax = axes[1, 0]
versions = ['v1.0', 'v2.0', 'v2.1-canary']
traffic = [60, 35, 5]
colors = ['blue', 'green', 'orange']

ax.pie(traffic, labels=versions, colors=colors, autopct='%.1f%%')
ax.set_title('Traffic Distribution by Model Version')

# 5. Resource utilization
ax = axes[1, 1]
cpu = 40 + 20 * np.sin(time_points / 15) + np.random.normal(0, 5, 100)
memory = 60 + 10 * np.sin(time_points / 20) + np.random.normal(0, 3,
100)

ax.plot(time_points, cpu, 'b-', label='CPU %', linewidth=2)
ax.plot(time_points, memory, 'g-', label='Memory %', linewidth=2)
ax.axhline(y=80, color='red', linestyle='--', label='Scale threshold')
ax.set_xlabel('Time (minutes)')
ax.set_ylabel('Utilization (%)')
ax.set_title('Resource Utilization')
ax.legend()
ax.grid(True, alpha=0.3)

# 6. A/B test results
ax = axes[1, 2]
days = np.arange(7)
model_a_accuracy = [0.85, 0.84, 0.86, 0.85, 0.84, 0.85, 0.86]
model_b_accuracy = [0.87, 0.88, 0.89, 0.88, 0.89, 0.90, 0.91]

x = np.arange(len(days))
width = 0.35

ax.bar(x - width/2, model_a_accuracy, width, label='Model A',
alpha=0.8)
ax.bar(x + width/2, model_b_accuracy, width, label='Model B',
alpha=0.8)
ax.set_xlabel('Day')
ax.set_ylabel('Accuracy')
ax.set_title('A/B Test: Model Performance')
ax.set_xticks(x)
ax.set_xticklabels(days)
ax.legend()
```

```
ax.grid(True, alpha=0.3, axis='y')

plt.suptitle('Production Monitoring Dashboard', fontsize=16)
plt.tight_layout()
plt.show()

# Test production API
def test_production_api():
    """Test the production API with simulated traffic"""

    # Create API instance
    api = ProductionMLAPI()

    # Deploy models
    api.models['v1.0'] = "model_v1" # Dummy model
    api.models['v2.0'] = "model_v2" # Dummy model
    api.model_versions['v1.0'] = {'deployed_at':
        datetime.now().isoformat()}
    api.model_versions['v2.0'] = {'deployed_at':
        datetime.now().isoformat()}

    # Create A/B test
    api.ab_tests['test1'] = {
        'model_a': 'v1.0',
        'model_b': 'v2.0',
        'traffic_split': 0.5
    }

    # Simulate requests
    print("Simulating API requests...")

    with api.app.test_client() as client:
        # Health check
        response = client.get('/health')
        print(f"Health check: {response.get_json()}")

        # Make predictions
        for i in range(10):
            features = np.random.randn(20).tolist()

            response = client.post('/predict',
                json={
                    'features': features,
                    'user_id': f'user_{i}'
                })
            if response.status_code == 200:
                result = response.get_json()
                print(f"Prediction {i}: {result['prediction']:.4f}, "
                      f"Model: {result['model_version']}"), 514
```

```
f"Latency: {result['latency_ms']:.2f}ms")\n\n# Get monitoring metrics\nresponse = client.get('/monitoring/dashboard')\nmetrics = response.get_json()\nprint(f"\nMonitoring Metrics: {metrics}")\n\n# Create deployment configurations\nconfigs = create_deployment_configurations()\n\n# Save deployment files\nwith open('Dockerfile', 'w') as f:\nf.write(configs['dockerfile'])\n\nwith open('k8s-deployment.yaml', 'w') as f:\nf.write(configs['k8s_deployment'])\n\nwith open('docker-compose.yml', 'w') as f:\nf.write(configs['docker_compose'])\n\nwith open('requirements.txt', 'w') as f:\nf.write(configs['requirements'])\n\nprint("\nDeployment files created!")\n\n# Visualize monitoring\nvisualize_production_monitoring()\n\nprint("\nProduction API test complete!")\n\nif __name__ == "__main__":\ntest_production_api()
```

**Task:** Design a system that trains models across distributed data sources.

```
import numpy as np
from typing import List, Dict, Tuple
import hashlib
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification

class FederatedLearningSystem:
    """
        Federated learning system with privacy preservation
    """

    def __init__(self, n_clients, model_fn,
                 aggregation_strategy='fedavg'):
        self.n_clients = n_clients
        self.model_fn = model_fn
        self.aggregation_strategy = aggregation_strategy
        self.global_model = None
        self.client_models = {}
        self.client_data = {}
        self.training_history = []
        self.privacy_budget = 1.0

    def distribute_data(self, X, y, distribution='iid'):
        """Distribute data among clients"""
        n_samples = len(X)

        if distribution == 'iid':
            # IID distribution
            indices = np.random.permutation(n_samples)
            samples_per_client = n_samples // self.n_clients

            for i in range(self.n_clients):
                start_idx = i * samples_per_client
                end_idx = start_idx + samples_per_client
                if i == self.n_clients - 1: # Last client gets remaining
samples
                    end_idx = n_samples

                client_indices = indices[start_idx:end_idx]
                self.client_data[i] = {
                    'X': X[client_indices],
                    'y': y[client_indices]
                }

    def train_global_model(self):
        # Implement global model training logic here
        pass

    def evaluate_global_model(self):
        # Implement global model evaluation logic here
        pass
```

```

        elif distribution == 'non_iid':
            # Non-IID distribution (by class)
            unique_classes = np.unique(y)
            n_classes_per_client = max(2, len(unique_classes) // self.n_clients)

            # Assign classes to clients
            class_assignment = {}
            for i in range(self.n_clients):
                # Each client gets a subset of classes
                client_classes = np.random.choice(
                    unique_classes,
                    min(n_classes_per_client, len(unique_classes)),
                    replace=False
                )
                class_assignment[i] = client_classes

            # Distribute data
            for i in range(self.n_clients):
                client_classes = class_assignment[i]
                client_indices = np.where(np.isin(y, client_classes))[0]

                # Limit samples per client
                if len(client_indices) > n_samples // self.n_clients:
                    client_indices = np.random.choice(
                        client_indices,
                        n_samples // self.n_clients,
                        replace=False
                    )

                self.client_data[i] = {
                    'X': X[client_indices],
                    'y': y[client_indices]
                }

            # Print distribution statistics
            print(f"\nData Distribution ({distribution}):")
            for i in range(min(5, self.n_clients)): # Show first 5 clients
                client_y = self.client_data[i]['y']
                unique, counts = np.unique(client_y, return_counts=True)
                print(f"Client {i}: {len(client_y)} samples, classes: {dict(zip(unique, counts))}")

        def initialize_global_model(self, input_shape, output_shape):
            """Initialize the global model"""
            self.global_model = self.model_fn(input_shape, output_shape)

            # Initialize client models
            for i in range(self.n_clients):
                self.client_models[i] = self.model_fn(input_shape,

```

```

output_shape)

def train_client(self, client_id, epochs=1, privacy_noise=None):
    """Train model on client's local data"""
    # Get client data
    X_client = self.client_data[client_id]['X']
    y_client = self.client_data[client_id]['y']

    # Get current global weights
    global_weights = self.global_model.get_weights()

    # Set client model to global weights
    self.client_models[client_id].set_weights(global_weights)

    # Train locally
    history = self.client_models[client_id].fit(
        X_client, y_client,
        epochs=epochs,
        batch_size=32,
        verbose=0
    )

    # Add differential privacy noise if specified
    if privacy_noise is not None:
        client_weights = self.client_models[client_id].get_weights()
        noisy_weights = []

        for w in client_weights:
            noise = np.random.laplace(0, privacy_noise, w.shape)
            noisy_weights.append(w + noise)

        self.client_models[client_id].set_weights(noisy_weights)

        # Update privacy budget
        self.privacy_budget -= privacy_noise * epochs

    return history.history['loss'][-1]

def aggregate_models(self, selected_clients=None):
    """Aggregate client models using specified strategy"""
    if selected_clients is None:
        selected_clients = list(range(self.n_clients))

    if self.aggregation_strategy == 'fedavg':
        # Federated Averaging
        aggregated_weights =
self._federated_averaging(selected_clients)
    elif self.aggregation_strategy == 'median':
        # Median aggregation (robust to outliers)
        aggregated_weights =

```

```

self._median_aggregation(selected_clients)
    elif self.aggregation_strategy == 'trimmed_mean':
        # Trimmed mean (robust to poisoning)
        aggregated_weights =
self._trimmed_mean_aggregation(selected_clients)
    else:
        raise ValueError(f"Unknown aggregation strategy:
{self.aggregation_strategy}")

# Update global model
self.global_model.set_weights(aggregated_weights)

def _federated_averaging(self, selected_clients):
    """Standard federated averaging"""
    # Get weights from selected clients
    client_weights = []
    client_sizes = []

    for client_id in selected_clients:
        weights = self.client_models[client_id].get_weights()
        client_weights.append(weights)
        client_sizes.append(len(self.client_data[client_id]['y']))

    # Weighted average based on data size
    total_size = sum(client_sizes)
    aggregated_weights = []

    for layer_idx in range(len(client_weights[0])):
        layer_weights = []

        for client_idx, client_id in enumerate(selected_clients):
            weight = client_sizes[client_idx] / total_size
            layer_weights.append(client_weights[client_idx][layer_idx]
* weight)

        aggregated_weights.append(sum(layer_weights))

    return aggregated_weights

def _median_aggregation(self, selected_clients):
    """Median aggregation for robustness"""
    client_weights = []

    for client_id in selected_clients:
        weights = self.client_models[client_id].get_weights()
        client_weights.append(weights)

    aggregated_weights = []

    for layer_idx in range(len(client_weights[0])):

```

```
# Stack weights from all clients
stacked_weights = np.stack([
    client_weights[i][layer_idx]
    for i in range(len(selected_clients))
])

# Take median across clients
median_weight = np.median(stacked_weights, axis=0)
aggregated_weights.append(median_weight)

return aggregated_weights

def _trimmed_mean_aggregation(self, selected_clients, trim_ratio=0.1):
    """Trimmed mean aggregation"""
    client_weights = []

    for client_id in selected_clients:
        weights = self.client_models[client_id].get_weights()
        client_weights.append(weights)

    aggregated_weights = []

    for layer_idx in range(len(client_weights[0])):
        # Stack weights from all clients
        stacked_weights = np.stack([
            client_weights[i][layer_idx]
            for i in range(len(selected_clients))
        ])

        # Sort and trim
        n_trim = int(len(selected_clients) * trim_ratio)
        if n_trim > 0:
            # Sort along client axis
            sorted_weights = np.sort(stacked_weights, axis=0)
            trimmed_weights = sorted_weights[n_trim:-n_trim]
        else:
            trimmed_weights = stacked_weights

        # Take mean of trimmed weights
        mean_weight = np.mean(trimmed_weights, axis=0)
        aggregated_weights.append(mean_weight)

    return aggregated_weights

def detect_poisoning_attacks(self, threshold=2.0):
    """Detect potential poisoning attacks"""
    suspicious_clients = []

    # Get global model weights
    global_weights = self.global_model.get_weights()
```

```
for client_id in range(self.n_clients):
    if client_id not in self.client_models:
        continue

    client_weights = self.client_models[client_id].get_weights()

    # Calculate distance from global model
    distances = []
    for g_w, c_w in zip(global_weights, client_weights):
        dist = np.linalg.norm(g_w - c_w)
        distances.append(dist)

    total_distance = sum(distances)

    # Check if distance is anomalous
    if total_distance > threshold:
        suspicious_clients.append({
            'client_id': client_id,
            'distance': total_distance
        })

return suspicious_clients

def secure_aggregation(self, selected_clients):      521
    """Secure aggregation with encryption (simplified)"""
    # Simplified secure aggregation using secret sharing

    # Each client creates random masks
    client_masks = {}
    for client_id in selected_clients:
        mask = [np.random.randn(*w.shape) for w in
                self.client_models[client_id].get_weights()]
        client_masks[client_id] = mask

    # Share masks between clients (simplified)
    shared_masks = {}
    for i, client_i in enumerate(selected_clients):
        for j, client_j in enumerate(selected_clients):
            if i < j:
                # Create pairwise mask
                seed = hashlib.sha256(
                    f"{client_i}-{client_j}".encode()
                ).hexdigest()
                np.random.seed(int(seed[:8], 16))

                pairwise_mask = [
                    np.random.randn(*w.shape)
                    for w in
self.client_models[client_i].get_weights()
```

```

        ]

        shared_masks[(client_i, client_j)] = pairwise_mask

    # Aggregate masked weights
    masked_weights = []

    for client_id in selected_clients:
        weights = self.client_models[client_id].get_weights()

        # Add client's mask
        masked = []
        for w, m in zip(weights, client_masks[client_id]):
            masked.append(w + m)

        # Add/subtract pairwise masks
        for (i, j), mask in shared_masks.items():
            if client_id == i:
                for idx, m in enumerate(mask):
                    masked[idx] += m
            elif client_id == j:
                for idx, m in enumerate(mask):
                    masked[idx] -= m

        masked_weights.append(masked)

    # Aggregate (masks cancel out)
    aggregated = self._federated_averaging(selected_clients)

    return aggregated

def train_federated(self, rounds=10, clients_per_round=None,
                    local_epochs=1, privacy_noise=None):
    """Main federated training loop"""
    if clients_per_round is None:
        clients_per_round = self.n_clients

    for round_num in range(rounds):
        print(f"\n--- Round {round_num + 1}/{rounds} ---")

        # Select clients for this round
        selected_clients = np.random.choice(
            self.n_clients,
            min(clients_per_round, self.n_clients),
            replace=False
        )

        # Train selected clients
        client_losses = []
        for client_id in selected_clients:

```

```

        loss = self.train_client(
            client_id,
            epochs=local_epochs,
            privacy_noise=privacy_noise
        )
        client_losses.append(loss)
        print(f"Client {client_id} loss: {loss:.4f}")

    # Aggregate models
    if privacy_noise is None:
        self.aggregate_models(selected_clients)
    else:
        # Use secure aggregation for privacy
        aggregated_weights =
self.secure_aggregation(selected_clients)
        self.global_model.set_weights(aggregated_weights)

    # Detect poisoning attempts
    suspicious = self.detect_poisoning_attacks()
    if suspicious:
        print(f"Warning: Suspicious clients detected:
{suspicious}")

    # Evaluate global model
    global_loss, global_acc = self.evaluate_global_model() 523

    # Store history
    self.training_history.append({
        'round': round_num,
        'client_losses': client_losses,
        'global_loss': global_loss,
        'global_accuracy': global_acc,
        'suspicious_clients': suspicious,
        'privacy_budget': self.privacy_budget
    })

    print(f"Global model - Loss: {global_loss:.4f}, Accuracy:
{global_acc:.4f}")
    print(f"Privacy budget remaining: {self.privacy_budget:.4f}")

def evaluate_global_model(self):
    """Evaluate global model on all client data"""
    all_losses = []
    all_accuracies = []

    for client_id in range(self.n_clients):
        if client_id not in self.client_data:
            continue

        X = self.client_data[client_id]['X']

```

```

y = self.client_data[client_id]['y']

loss, acc = self.global_model.evaluate(X, y, verbose=0)
all_losses.append(loss)
all_accuracies.append(acc)

return np.mean(all_losses), np.mean(all_accuracies)

def visualize_federated_training(self):
    """Visualize federated training process"""

fig, axes = plt.subplots(2, 3, figsize=(15, 10))

# 1. Global model performance
ax = axes[0, 0]

rounds = [h['round'] for h in self.training_history]
global_losses = [h['global_loss'] for h in self.training_history]
global_accs = [h['global_accuracy'] for h in
self.training_history]

ax2 = ax.twinx()

ax.plot(rounds, global_losses, 'b-', linewidth=2, label='Loss')
ax2.plot(rounds, global_accs, 'r-', linewidth=2, label='Accuracy') 524

ax.set_xlabel('Round')
ax.set_ylabel('Loss', color='b')
ax2.set_ylabel('Accuracy', color='r')
ax.set_title('Global Model Performance')
ax.grid(True, alpha=0.3)

# 2. Client loss distribution
ax = axes[0, 1]

all_client_losses = []
for h in self.training_history:
    all_client_losses.extend(h['client_losses'])

ax.hist(all_client_losses, bins=30, alpha=0.7, color='green')
ax.set_xlabel('Client Loss')
ax.set_ylabel('Frequency')
ax.set_title('Client Loss Distribution')
ax.grid(True, alpha=0.3)

# 3. Data distribution
ax = axes[0, 2]

client_sizes = []
client_class_counts = []

```

```

for i in range(min(10, self.n_clients)): # First 10 clients
    if i in self.client_data:
        size = len(self.client_data[i]['y'])
        n_classes = len(np.unique(self.client_data[i]['y']))
        client_sizes.append(size)
        client_class_counts.append(n_classes)

x = np.arange(len(client_sizes))
width = 0.35

ax.bar(x - width/2, client_sizes, width, label='Samples',
alpha=0.8)
ax.bar(x + width/2, np.array(client_class_counts) * 20, width,
label='Classes (×20)', alpha=0.8)

ax.set_xlabel('Client ID')
ax.set_ylabel('Count')
ax.set_title('Client Data Distribution')
ax.legend()
ax.grid(True, alpha=0.3, axis='y')

# 4. Privacy budget
ax = axes[1, 0]

privacy_budgets = [h['privacy_budget'] for h in
self.training_history]

ax.plot(rounds, privacy_budgets, 'r-', linewidth=2)
ax.axhline(y=0, color='black', linestyle='--', label='Depleted')
ax.set_xlabel('Round')
ax.set_ylabel('Privacy Budget ( $\epsilon$ )')
ax.set_title('Differential Privacy Budget')
ax.legend()
ax.grid(True, alpha=0.3)

# 5. Attack detection
ax = axes[1, 1]

suspicious_counts = []
for h in self.training_history:
    suspicious_counts.append(len(h.get('suspicious_clients', [])))

ax.bar(rounds, suspicious_counts, color='red', alpha=0.7)
ax.set_xlabel('Round')
ax.set_ylabel('Suspicious Clients')
ax.set_title('Potential Attack Detection')
ax.grid(True, alpha=0.3, axis='y')

# 6. Convergence analysis

```

```

ax = axes[1, 2]

if len(global_losses) > 1:
    loss_changes = np.diff(global_losses)
    ax.plot(rounds[1:], loss_changes, 'b-', linewidth=2)
    ax.axhline(y=0, color='black', linestyle='--')
    ax.set_xlabel('Round')
    ax.set_ylabel('Loss Change')
    ax.set_title('Convergence Rate')
    ax.grid(True, alpha=0.3)

plt.suptitle('Federated Learning System Analysis', fontsize=16)
plt.tight_layout()
plt.show()

# Test federated learning system
def test_federated_learning():
    # Generate dataset
    X, y = make_classification(
        n_samples=10000,
        n_features=20,
        n_informative=15,
        n_classes=5,
        random_state=42
    )

    # Define model function
    def create_model(input_shape, output_shape):
        model = keras.Sequential([
            layers.Dense(64, activation='relu', input_shape=(input_shape,)),
            layers.BatchNormalization(),
            layers.Dropout(0.3),
            layers.Dense(32, activation='relu'),
            layers.BatchNormalization(),
            layers.Dense(output_shape, activation='softmax')
        ])

        model.compile(
            optimizer='adam',
            loss='sparse_categorical_crossentropy',
            metrics=['accuracy']
        )

        return model

    # Create federated system
    fed_system = FederatedLearningSystem(
        n_clients=10,
        model_fn=create_model,

```

```

        aggregation_strategy='fedavg'
    )

# Distribute data (non-IID)
fed_system.distribute_data(X, y, distribution='non_iid')

# Initialize global model
fed_system.initialize_global_model(
    input_shape=X.shape[1],
    output_shape=len(np.unique(y))
)

# Train with federated learning
print("Starting federated training...")
fed_system.train_federated(
    rounds=20,
    clients_per_round=5,
    local_epochs=2,
    privacy_noise=0.01 # Differential privacy
)

# Visualize results
fed_system.visualize_federated_training()

# Test poisoning detection
print("\n==== Testing Poisoning Detection ====")

# Simulate poisoned client
poisoned_client = 0
if poisoned_client in fed_system.client_models:
    poisoned_weights =
fed_system.client_models[poisoned_client].get_weights()

    # Add large perturbation
    for i in range(len(poisoned_weights)):
        poisoned_weights[i] +=
np.random.randn(*poisoned_weights[i].shape) * 5

fed_system.client_models[poisoned_client].set_weights(poisoned_weights)

    # Detect
    suspicious = fed_system.detect_poisoning_attacks()
    print(f"Detected suspicious clients: {suspicious}")

# Test different aggregation strategies
print("\n==== Testing Aggregation Strategies ====")

strategies = ['fedavg', 'median', 'trimmed_mean']
results = {}

```

```
for strategy in strategies:
    print(f"\nTesting {strategy} aggregation...")

    # Create new system
    test_system = FederatedLearningSystem(
        n_clients=10,
        model_fn=create_model,
        aggregation_strategy=strategy
    )

    # Use same data distribution
    test_system.client_data = fed_system.client_data

    # Initialize
    test_system.initialize_global_model(
        input_shape=X.shape[1],
        output_shape=len(np.unique(y))
    )

    # Train briefly
    test_system.train_federated(
        rounds=5,
        clients_per_round=5,
        local_epochs=1
    )

    # Evaluate
    loss, acc = test_system.evaluate_global_model()
    results[strategy] = {'loss': loss, 'accuracy': acc}

    print(f"{strategy}: Loss={loss:.4f}, Accuracy={acc:.4f}")

print("\n==== Federated Learning Test Complete ====")

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

528

# Index

- Activation function: 160
- Aggregation Pipeline: 43
- Algorithm Selection: 5, 16, 137, 156
- Algorithm Selection Matrix: 5
- Anomaly Detection: 9, 22, 28, 69, 91, 98, 113, 126, 136, 137, 145, 151, 153, 155, 156, 177, 179, 180, 190, 202, 206, 210, 279, 280, 285, 295, 300, 305, 311, 314, 355, 361, 362, 391, 399, 407, 409
- Architecture Selection: 16, 210
- Arrays: 33, 396
- Arrays vs Lists: 33
- Autoencoder Loss Functions: 9
- BSON: 42
- Bagging: 132
- Bayes' Theorem: 3, 69, 79, 82, 87, 114
- Beaconing detection: 91, 99
- Bernoulli: 7, 80, 179
- Binomial: 3, 80, 81, 84, 276
- CNN: 9, 16, 22, 183, 184, 186, 192, 193, 194, 195, 208, 210, 214, 218, 371, 372, 373, 376, 377, 421
- CNN Architecture Guidelines: 9
- CTEs: 38
- Clustering: 5, 16, 17, 114, 115, 116, 117, 137, 140, 141, 143, 155, 156, 295, 297, 298, 301, 388, 389, 390
- Comparison Operators: 43
- Conditional: 3, 81, 114
- Continuous Distributions: 80
- Convolution: 9, 180, 181, 182, 184, 196, 210
- Convolutional Neural Networks: 179, 180
- Cosine: 7, 89, 117, 350
- DBSCAN: 5, 18, 57, 115, 116, 123, 124, 136, 137, 141, 142, 143, 151, 153, 155, 156, 295, 298, 299, 300, 301, 303
- DBScan vs K-Means: 123
- DCL: 37
- DDL: 37, 39
- DDoS Pattern Recognition: 91
- DFT: 89, 91

- DML: 37, 39
- DQL: 37, 40
- Data Acquisition: 1, 32, 243, 244
- Data Cleaning: 1, 32, 59, 63, 66, 68
- Data Cleaning Pipeline: 1, 59, 63
- Data Exfiltration: 91, 138, 296
- Data Extraction: 37, 68
- Data Manipulation: 37
- Data Structures: 32, 34, 35, 68, 399
- Data aggregation: 37
- Data filtering: 37
- Data joining: 37
- Decision Trees: 115, 129, 131, 132
- Denoising: 190, 198, 199, 208, 379, 380, 381, 382, 384, 385, 386
- Descriptive Statistics: 3, 114
- Distributions Quick Reference: 3
- ELU: 160, 230, 342, 349, 364, 444
- Elbow: 5, 18, 118, 126, 140, 142, 298
- Embedding: 10, 22, 180, 188, 189, 195, 196, 210, 320, 323, 399
- Entropy: 5, 7, 28, 129, 162, 163, 173, 179, 390, 401
- Essential NumPy Functions: 33
- FDR: 103
- FFT: 91, 92, 93, 94, 96, 99, 113, 114
- False Discovery Rate: 103
- Federated Learning System: 240, 471, 481
- Forests: 115, 156
- Fourier Series: 89
- Frequency domain: 89, 93
- Gaussian: 72, 85, 86, 116, 128, 208, 289, 290, 299, 379, 382, 386
- Geometric: 13, 73, 85, 223, 243
- Gini: 5, 129, 130
- Grow tree: 132
- HDBSCAN: 116
- Hinge: 162
- IQR: 1, 2, 3, 57, 58, 62, 64, 71, 72, 73, 74, 75, 76, 114, 279, 280, 281
- Indexes: 1, 39
- Information Gain: 129, 130
- JSON: 24, 42, 44, 45, 46, 244, 248, 249, 251, 253, 254, 255, 256, 257, 258, 259, 260, 264, 269, 271, 456, 469

- K Value: 121
- K-Means: 5, 17, 115, 116, 118, 120, 123, 124, 137, 140, 141, 153, 155, 156, 297, 300
- K-Medoids: 116, 118, 120
- K-Modes: 116, 120
- KNN: 121, 156, 436
- Kernel Functions: 5, 127
- Kernel Trick: 127
- Kurtosis: 72, 74, 76
- L2: 7, 26, 169, 170, 171, 177, 349, 444, 445
- Latent dimension: 200, 210, 394
- Leaky ReLU: 7, 160
- Lists: 33, 253
- Logical Operators: 43
- Loss Functions: 7, 9, 162, 173, 179, 240
- MAD: 3, 57, 62, 71, 72, 73, 74, 75, 76, 77
- MAE: 9, 15, 16, 162, 163, 177, 191, 283, 284, 338, 339, 340, 341, 343, 380, 389, 445, 451
- MAR: 56, 57
- MCAR: 56, 57
- MNAR: 56, 57, 66
- MSE: 7, 9, 15, 22, 129, 162, 163, 177, 190, 191, 199, 203, 283, 284, 306, 307, 338, 339, 340, 341, 343, 356, 380, 389, 414, 445
- Machine Learning: 32, 68, 69, 114, 115, 156, 157, 180, 211, 243, 311, 325
- Metric Selection: 16
- Model Training: 15
- MongoDB: 42, 43, 44, 45, 46, 66, 68
- MongoDB query operators: 43
- Neural Network Formulas: 7
- NoSQL: 32, 42, 68
- NumPy: 1, 17, 24, 31, 33, 35, 59, 68, 72, 81, 91, 104, 137, 159, 166, 183, 192, 214, 234, 260, 272, 286, 295, 305, 317, 327, 338, 352, 411, 412, 432, 436, 444, 456, 461, 466, 471
- NumPy Essential Operations: 1
- OPTICS: 116
- PCA: 115, 125, 126, 137, 141, 142, 143, 144, 145, 151, 153, 156, 295, 301, 302, 317, 321, 394, 445, 446
- PCA Properties: 125
- PCA for anomaly detection: 126, 145
- Pearson: 102
- Perceptron: 158

- Performance: 12, 15, 35, 39, 46, 68, 91, 112, 124, 150, 176, 177, 178, 205, 208, 225, 234, 240, 283, 302, 305, 308, 309, 310, 311, 314, 325, 353, 365, 366, 367, 370, 375, 377, 382, 384, 393, 396, 418, 419, 427, 429, 432, 437, 438, 440, 452, 454, 468, 479
- Point Types: 123
- Poisson: 3, 80, 83, 99, 273, 274, 276, 277
- Probability Rules: 3, 78
- Python: 22, 25, 32, 33, 34, 68, 311, 463, 464, 466
- Python Data Structures: 32, 34
- Query Optimization: 38
- ReLU: 7, 9, 21, 22, 159, 160, 167, 169, 170, 174, 179, 193, 195, 197, 198, 219, 220, 221, 224, 230, 306, 339, 342, 349, 351, 355, 363, 364, 371, 372, 373, 379, 380, 388, 389, 391, 413, 421, 422, 440, 442, 444, 481
- Robust Measures: 3, 71, 73, 114
- SQL: 1, 32, 37, 38, 42, 43, 66, 68, 291, 292, 318
- SVM: 5, 20, 127, 128, 135, 137, 145, 146, 147, 151, 153, 155, 156, 162, 305, 307, 308
- SVM Key Concepts: 5
- Scaling Methods: 1, 57
- Scree: 126, 143
- Security Log Analysis: 35
- Set: 12, 18, 30, 34, 35, 37, 64, 78, 94, 134, 137, 142, 147, 203, 238, 254, 255, 256, 269, 286, 298, 299, 453, 459, 473
- Sigmoid: 7, 11, 22, 128, 159, 161, 163, 167, 170, 193, 195, 197, 198, 219, 220, 306, 356, 364, 423
- Silhouette: 119, 140, 297, 298, 299, 300, 301
- Softmax: 7, 11, 21, 159, 160, 161, 163, 173, 174, 179, 220, 226, 343, 349, 351, 363, 364, 372, 373, 388, 391, 400, 422, 423, 440, 442, 446, 481
- Spearman: 102
- Splitting Algorithm: 129
- Support Vector Machine: 19
- TCL: 37
- Tanh: 7, 128, 159, 230, 364, 444
- Time domain: 89, 93, 96
- Trees: 115, 129, 131, 132, 133, 134, 137, 150, 156
- Tuple: 34, 225, 226, 227, 471
- Web Scraping: 32, 48, 50, 68
- Window Functions: 37, 38, 40
- accuracy: 6, 11, 12, 16, 21, 23, 26, 56, 88, 134, 147, 148, 149, 150, 156, 170, 174, 175, 176, 193, 224, 226, 234, 240, 300, 309, 315, 340, 343, 350, 352, 353, 364, 366, 367, 368, 370, 373, 374, 375, 391, 400, 418, 419, 425, 429, 434, 438, 442, 451, 468, 478, 479, 481, 483

- aggregation: 28, 37, 40, 43, 45, 218, 473, 474, 475, 476, 478, 482, 483
- algorithm selection guide: 137
- alter: 37
- anova: 102, 107, 321
- autoencoder: 9, 22, 136, 155, 190, 197, 198, 199, 200, 202, 203, 208, 209, 210, 305, 306, 307, 308, 310, 311, 312, 356, 379, 380, 381, 382, 383, 384, 385, 388, 389, 395, 396, 399, 400, 401, 411, 412, 413, 414
- average: 23, 34, 37, 70, 71, 72, 83, 89, 119, 132, 134, 135, 183, 204, 229, 274, 358, 366, 375, 376, 393, 396, 407, 415, 417, 429, 435, 441, 474
- avg: 37, 40, 41, 45, 46, 132, 360, 361, 362, 418
- bias: 8, 56, 71, 127, 132, 158, 165, 344
- class imbalance: 26
- classifier: 85, 87, 88, 155, 174, 177, 286, 287, 293, 369
- commit: 40, 252, 256
- count: 1, 3, 18, 37, 40, 41, 46, 70, 80, 86, 121, 142, 237, 238, 248, 258, 273, 274, 275, 276, 277, 279, 280, 281, 282, 283, 284, 287, 288, 299, 325, 345, 360, 369, 395, 407, 453, 460, 480
- create: 33, 34, 37, 38, 39, 59, 62, 64, 66, 73, 75, 87, 88, 97, 98, 99, 108, 111, 113, 115, 129, 144, 155, 156, 169, 173, 177, 179, 184, 192, 194, 197, 202, 206, 208, 225, 228, 236, 237, 240, 252, 262, 278, 282, 297, 301, 314, 322, 323, 324, 325, 330, 331, 336, 342, 347, 348, 350, 354, 361, 363, 370, 372, 376, 377, 379, 383, 385, 386, 397, 398, 408, 409, 411, 417, 420, 423, 426, 429, 430, 432, 440, 441, 445, 446, 447, 449, 455, 459, 463, 467, 469, 470, 476, 481, 483
- cross-entropy: 7, 162, 173, 179, 401
- data quality dimensions: 56
- delete: 37, 56, 57, 262, 286, 291, 408, 409
- drop: 1, 7, 37, 88, 234, 282, 286, 291, 292, 293, 303, 322, 336
- dropout: 7, 21, 22, 26, 167, 169, 170, 171, 174, 177, 179, 193, 195, 208, 219, 220, 226, 230, 339, 343, 348, 349, 363, 364, 372, 373, 379, 380, 386, 389, 413, 421, 422, 432, 441, 442, 446, 481
- feature scaling methods: 57
- gap: 119, 353
- group\_concat: 37
- handling missing data: 56
- huber: 163, 380, 445
- insert: 37, 39, 40, 256, 286, 291
- isolation: 57, 135, 136, 137, 151, 155, 280, 281, 307
- joint: 78, 81, 114
- kruskal-wallis: 102, 107
- lazy: 121

- mann-whitney U: 106, 107
- margin-based: 156, 162
- marginal: 78
- matplotlib: 59, 72, 81, 91, 104, 137, 159, 166, 183, 192, 196, 205, 214, 234, 272, 295, 315, 323, 327, 338, 358, 404, 432, 444, 456, 466, 471
- max: 2, 7, 11, 24, 28, 37, 40, 45, 53, 58, 71, 75, 86, 97, 105, 119, 146, 149, 159, 160, 162, 163, 183, 184, 196, 204, 210, 226, 227, 248, 252, 256, 266, 276, 284, 286, 287, 289, 290, 302, 303, 306, 307, 309, 310, 319, 321, 334, 340, 344, 366, 369, 371, 392, 407, 433, 441, 451, 460, 461, 472
- mean: 3, 17, 22, 23, 28, 57, 70, 71, 72, 73, 75, 77, 81, 82, 84, 85, 86, 88, 92, 94, 97, 98, 104, 105, 106, 110, 114, 118, 134, 138, 139, 141, 142, 152, 162, 163, 183, 203, 204, 210, 235, 237, 274, 275, 276, 277, 281, 282, 284, 286, 288, 289, 290, 295, 296, 300, 304, 307, 308, 309, 310, 311, 312, 319, 320, 321, 324, 329, 335, 341, 344, 356, 357, 358, 366, 368, 369, 370, 375, 376, 382, 384, 386, 391, 392, 393, 395, 403, 414, 415, 416, 418, 419, 425, 426, 429, 435, 436, 437, 441, 451, 460, 474, 475, 479
- median: 2, 3, 57, 58, 62, 70, 71, 72, 73, 74, 75, 77, 114, 284, 415, 473, 474, 475, 482
- min: 2, 37, 45, 58, 62, 71, 75, 97, 105, 123, 133, 135, 139, 146, 168, 194, 199, 227, 238, 248, 266, 284, 307, 310, 319, 332, 334, 340, 344, 357, 376, 390, 393, 433, 441, 462, 472, 477, 480
- mode: 3, 17, 57, 70, 71, 72, 73, 114, 120, 327
- no pruning: 132
- normalization: 7, 169, 179, 207, 342, 349, 371, 445
- null hypothesis: 100
- odds ratio: 102
- outlier detection: 57, 66, 300
- overfitting: 14, 26, 30, 131, 133, 134, 148, 149, 157, 169, 171, 172, 179
- p-value: 100, 102, 104, 106, 108, 109, 275, 277
- pandas: 17, 31, 35, 39, 50, 59, 68, 72, 81, 91, 104, 137, 244, 260, 272, 286, 295, 305, 317, 327
- pymongo: 44
- pyplot: 59, 72, 81, 91, 104, 137, 159, 166, 183, 192, 214, 234, 272, 295, 315, 323, 327, 338, 432, 444, 456, 471
- rank: 38, 102, 105
- regression: 16, 127, 129, 132, 161, 162, 163, 177, 212, 220
- revoke: 37
- rollback: 14, 37, 176, 232
- rule-based: 156
- select: 1, 37, 38, 40, 41, 44, 49, 52, 53, 63, 226, 255, 257, 258, 274, 286, 291, 321, 322, 331, 332, 477
- signal analysis: 91, 97, 113, 114
- significance level: 100, 111

- skewness: 72, 74, 76
- sqlite3: 39, 251, 252, 255, 256, 257, 258, 260
- standard deviation: 71, 72, 114
- sum: 1, 5, 28, 37, 40, 45, 46, 59, 60, 61, 70, 87, 89, 118, 140, 145, 151, 152, 160, 173, 204, 205, 257, 274, 280, 281, 282, 284, 287, 288, 289, 299, 308, 309, 324, 358, 365, 366, 393, 395, 401, 402, 404, 407, 451, 460, 474, 476
- t-test: 102, 105
- union: 3, 78, 286, 291
- update: 8, 37, 50, 79, 82, 118, 164, 165, 232, 255, 256, 286, 291, 292, 308, 309, 310, 314, 357, 359, 362, 365, 393, 402, 403, 405, 458, 459, 473, 474
- variance: 3, 57, 71, 72, 73, 102, 109, 110, 114, 118, 121, 125, 132, 138, 141, 143, 144, 344, 444, 445, 446
- voting: 5, 136, 178, 202, 203, 204, 209, 210, 366, 415
- web scraping: 32, 48, 50, 68