Machine Learning Integration for YYSFold

From Statistical Methods to Deep Learning

YYSFold Research

December 5, 2025

Abstract

This document outlines how machine learning can be integrated into the YYSFold blockchain fingerprinting system. We analyze the current ML-adjacent techniques, identify opportunities for learned components, and provide a detailed implementation roadmap. Key enhancements include learned codebook training, neural anomaly detection, time-series prediction models, and graph neural networks for hypergraph analysis. We also address the unique challenge of maintaining zero-knowledge proof compatibility with learned models.

Contents

1 Current State: ML-Adjacent Techniques

YYSFold already employs several techniques from the broader ML/statistics ecosystem, though none currently use learned parameters from data.

1.1 Technique Inventory

Component	Technique	Type	Learned?
Product Quantization	Vector quantization	Unsupervised	No (random init)
KDE (Hotzones)	Kernel Density Est.	Non-parametric	No (fixed bandwidth)
Anomaly Score	Weighted heuristic	Rule-based	No (hand-tuned)
Predictions	Trend extrapolation	Rule-based	No (fixed rules)
Semantic Tags	Threshold classification	Rule-based	No (hand-tuned)

1.2 The Codebook Problem

The Product Quantization codebook is the most glaring opportunity. Currently:

```
// folding/codebook.ts - CURRENT: Random initialization
const centroids = Array.from({ length: numSubspaces }, () =>
  Array.from({ length: numCentroids }, () =>
    Array.from({ length: subvectorDim }, () => (rand() * 2 - 1) * scale
      ),
  ),
);
```

Problem: Random centroids don't capture the true distribution of blockchain transaction vectors. This leads to:

- Higher reconstruction error (PQ residuals)
- Less meaningful quantization codes
- Suboptimal compression for the actual data distribution

1.3 Mathematical Formulation of Current PQ

Given a vector $\mathbf{v} \in \mathbb{R}^d$, we split it into m subvectors:

$$\mathbf{v} = [\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_m], \quad \mathbf{v}_j \in \mathbb{R}^{d/m}$$

Each subvector is quantized to its nearest centroid from a codebook $\mathcal{C}_j$:

$$q_j(\mathbf{v}_j) = \arg\min_{c \in \mathcal{C}_j} \|\mathbf{v}_j - c\|_2$$

The reconstruction is:

$$\hat{\mathbf{v}} = [\mathcal{C}_1[q_1], \mathcal{C}_2[q_2], \ldots, \mathcal{C}_m[q_m]]$$

And the residual measures compression quality:

$$r = \|\mathbf{v} - \hat{\mathbf{v}}\|_2$$

Current issue: With random $\mathcal{C}_j$, the expected residual $\mathbb{E}[r]$ is unnecessarily high.

2 ML Enhancement #1: Learned PQ Codebook

2.1 Objective

Train codebook centroids using k-means clustering on historical block vectors, minimizing reconstruction error.

2.2 Algorithm

Algorithm 1 Codebook Training via K-Means

Require: Training vectors $\{\mathbf{v}^{(i)}\}_{i=1}^{N}$, subspaces m, centroids per subspace k
Ensure: Trained codebook $\mathcal{C} = \{\mathcal{C}_1, \ldots, \mathcal{C}_m\}$

for $j = 1$ to m **do**
    Extract subvectors: $S_j = \{\mathbf{v}_j^{(i)}\}_{i=1}^{N}$
    Initialize centroids $\mathcal{C}_j$ randomly (k-means++)
    repeat
        Assign each $\mathbf{s} \in S_j$ to nearest centroid
        Update centroids as cluster means
    until convergence
end forreturn $\mathcal{C}$

2.3 Implementation

```
import numpy as np
from sklearn.cluster import KMeans
import json

def train_codebook(vectors: np.ndarray,
                   m: int = 4,
                   k: int = 256) -> dict:
    """
    Train PQ codebook using k-means.

    Args:
        vectors: (N, d) array of training vectors
        m: number of subspaces
        k: centroids per subspace

    Returns:
        Codebook dictionary with 'centroids' and 'metadata'
    """
    N, d = vectors.shape
    subvec_dim = d // m

    codebook = {
        'centroids': [],
        'metadata': {
            'num_subspaces': m,
            'num_centroids': k,
            'subvector_dim': subvec_dim,
            'training_samples': N
        }
    }

    for j in range(m):
```

```
        # Extract j-th subvector from all training vectors
        start = j * subvec_dim
        end = (j + 1) * subvec_dim
        subvectors = vectors[:, start:end]

        # Train k-means
        kmeans = KMeans(n_clusters=k,
                        init='k-means++',
                        n_init=10,
                        random_state=42)
        kmeans.fit(subvectors)

        # Store centroids
        codebook['centroids'].append(kmeans.cluster_centers_.tolist())

        # Log inertia (sum of squared distances)
        print(f"Subspace {j}: inertia = {kmeans.inertia_:.4f}")

    return codebook

# Usage
vectors = load_training_vectors()  # Load from artifacts/
codebook = train_codebook(vectors, m=4, k=256)
save_codebook(codebook, 'artifacts/learned-codebook.json')
```

2.4 Expected Improvement

Proposition 1. *A learned codebook reduces expected reconstruction error by a factor proportional to how well the training distribution matches the test distribution.*

For blockchain data with consistent patterns:

- **Random codebook:** $\mathbb{E}[r] \approx 5 - 8$ (current)
- **Learned codebook:** $\mathbb{E}[r] \approx 1 - 3$ (expected)

This directly improves the "PQ" metric shown on the dashboard.

3 ML Enhancement #2: Neural Anomaly Detection

3.1 Current Approach

The anomaly score is a weighted sum of heuristics:

$$\text{AnomalyScore} = w_1 \cdot \text{DensityFactor} + w_2 \cdot \text{ResidualFactor} + w_3 \cdot \text{TagFactor}$$

Where weights $(w_1, w_2, w_3) = (0.5, 0.35, 0.15)$ are hand-tuned.

3.2 Option A: Autoencoder Anomaly Detection

Learn a compressed representation of "normal" blocks. Anomalies have high reconstruction error.

```
import torch
import torch.nn as nn

class AnomalyAutoencoder(nn.Module):
    """
```

```
    Autoencoder for anomaly detection on block fingerprints.
    Anomaly score = reconstruction error.
    """
    def __init__(self, input_dim=96, hidden_dims=[64, 32, 16]):
        super().__init__()

        # Encoder
        encoder_layers = []
        prev_dim = input_dim
        for h in hidden_dims:
            encoder_layers.extend([
                nn.Linear(prev_dim, h),
                nn.ReLU(),
                nn.BatchNorm1d(h)
            ])
            prev_dim = h
        self.encoder = nn.Sequential(*encoder_layers)

        # Decoder (mirror of encoder)
        decoder_layers = []
        for h in reversed(hidden_dims[:-1]):
            decoder_layers.extend([
                nn.Linear(prev_dim, h),
                nn.ReLU(),
                nn.BatchNorm1d(h)
            ])
            prev_dim = h
        decoder_layers.append(nn.Linear(prev_dim, input_dim))
        self.decoder = nn.Sequential(*decoder_layers)

    def forward(self, x):
        z = self.encoder(x)
        return self.decoder(z)

    def anomaly_score(self, x):
        """Compute anomaly score as reconstruction error."""
        with torch.no_grad():
            x_hat = self.forward(x)
            return torch.mean((x - x_hat) ** 2, dim=-1)

# Training
def train_autoencoder(model, train_loader, epochs=100, lr=1e-3):
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)
    criterion = nn.MSELoss()

    for epoch in range(epochs):
        total_loss = 0
        for batch in train_loader:
            optimizer.zero_grad()
            x_hat = model(batch)
            loss = criterion(x_hat, batch)
            loss.backward()
            optimizer.step()
            total_loss += loss.item()

        if epoch % 10 == 0:
            print(f"Epoch {epoch}: Loss = {total_loss/len(train_loader)
                :.6f}")
```

```
    return model
```

3.3 Option B: Isolation Forest

A simpler, non-neural approach that isolates anomalies through random partitioning.

```
from sklearn.ensemble import IsolationForest
import numpy as np

class IsolationForestAnomalyDetector:
    def __init__(self, contamination=0.05):
        """
        contamination: expected proportion of anomalies
        """
        self.model = IsolationForest(
            n_estimators=200,
            contamination=contamination,
            random_state=42,
            n_jobs=-1
        )
        self.fitted = False

    def fit(self, fingerprints: np.ndarray):
        """Train on historical block fingerprints."""
        self.model.fit(fingerprints)
        self.fitted = True
        return self

    def score(self, fingerprint: np.ndarray) -> float:
        """
        Returns anomaly score in [0, 1].
        Higher = more anomalous.
        """
        if not self.fitted:
            raise ValueError("Model not fitted")

        # Isolation Forest returns negative scores
        # More negative = more anomalous
        raw_score = self.model.score_samples(fingerprint.reshape(1, -1)
            )[0]

        # Normalize to [0, 1] where 1 = most anomalous
        # Typical scores range from -0.7 (normal) to -1.0 (anomaly)
        normalized = np.clip((-raw_score - 0.5) * 2, 0, 1)
        return float(normalized)

# Usage
detector = IsolationForestAnomalyDetector(contamination=0.05)
detector.fit(historical_fingerprints)

# Score new block
new_fingerprint = compute_fingerprint(new_block)
anomaly = detector.score(new_fingerprint)
print(f"Anomaly score: {anomaly:.3f}")
```

3.4 Comparison

Method	Training Time	Inference	Interpretable	Data Needed
Current (heuristic)	None	Fast	Yes	None
Isolation Forest	Minutes	Fast	Partial	1000+ blocks
Autoencoder	Hours	Medium	No	10000+ blocks

Recommendation: Start with Isolation Forest for quick wins, then add Autoencoder for more nuanced detection.

4 ML Enhancement #3: Time Series Prediction

4.1 The Prediction Problem

Given a sequence of block fingerprints $\mathbf{f}_{t-k}, \ldots, \mathbf{f}_{t-1}$, predict the next fingerprint $\mathbf{f}_t$ and its associated tags.

Definition 1 (Block Prediction Task)**.** *Let $\mathcal{F} = \{\mathbf{f}_1, \mathbf{f}_2, \ldots\}$ be the sequence of block fingerprints and $\mathcal{T} = \{T_1, T_2, \ldots\}$ be their tag sets. The prediction task is:*

$$(\hat{\mathbf{f}}_t, \hat{T}_t) = g(\mathbf{f}_{t-k:t-1})$$

where g is the prediction model.

4.2 LSTM Predictor

```python
import torch
import torch.nn as nn

class BlockPredictor(nn.Module):
    """
    LSTM-based predictor for next block fingerprint and tags.
    """
    def __init__(self,
                 input_dim=96,       # Fingerprint dimension
                 hidden_dim=128,     # LSTM hidden size
                 num_layers=2,       # LSTM layers
                 num_tags=15,        # Number of possible tags
                 dropout=0.2):
        super().__init__()

        self.lstm = nn.LSTM(
            input_size=input_dim,
            hidden_size=hidden_dim,
            num_layers=num_layers,
            batch_first=True,
            dropout=dropout
        )

        # Fingerprint prediction head
        self.fingerprint_head = nn.Sequential(
            nn.Linear(hidden_dim, hidden_dim // 2),
            nn.ReLU(),
            nn.Linear(hidden_dim // 2, input_dim)
        )
```

```
        # Tag prediction head (multi-label)
        self.tag_head = nn.Sequential(
            nn.Linear(hidden_dim, hidden_dim // 2),
            nn.ReLU(),
            nn.Dropout(dropout),
            nn.Linear(hidden_dim // 2, num_tags),
            nn.Sigmoid()
        )

        # Confidence estimation
        self.confidence_head = nn.Sequential(
            nn.Linear(hidden_dim, 32),
            nn.ReLU(),
            nn.Linear(32, 1),
            nn.Sigmoid()
        )

    def forward(self, sequence):
        """
        Args:
            sequence: (batch, seq_len, input_dim) - history of
                fingerprints

        Returns:
            predicted_fingerprint: (batch, input_dim)
            predicted_tags: (batch, num_tags)
            confidence: (batch, 1)
        """
        # LSTM encoding
        lstm_out, (h_n, c_n) = self.lstm(sequence)

        # Use last hidden state
        last_hidden = lstm_out[:, -1, :]  # (batch, hidden_dim)

        # Predictions
        fingerprint = self.fingerprint_head(last_hidden)
        tags = self.tag_head(last_hidden)
        confidence = self.confidence_head(last_hidden)

        return fingerprint, tags, confidence

# Training loop
def train_predictor(model, train_loader, val_loader, epochs=50):
    optimizer = torch.optim.AdamW(model.parameters(), lr=1e-3)
    fingerprint_loss = nn.MSELoss()
    tag_loss = nn.BCELoss()

    for epoch in range(epochs):
        model.train()
        for batch in train_loader:
            sequences, target_fp, target_tags = batch

            optimizer.zero_grad()
            pred_fp, pred_tags, confidence = model(sequences)

            # Combined loss
            loss = (fingerprint_loss(pred_fp, target_fp) +
                    tag_loss(pred_tags, target_tags))
```

```
            loss.backward()
            optimizer.step()

        # Validation
        model.eval()
        val_acc = evaluate(model, val_loader)
        print(f"Epoch {epoch}: Val accuracy = {val_acc:.3f}")
```

4.3 Transformer Alternative

For longer context and better parallelization:

```
class TransformerPredictor(nn.Module):
    """
    Transformer-based predictor with attention over block history.
    """
    def __init__(self,
                 input_dim=96,
                 d_model=128,
                 nhead=4,
                 num_layers=3,
                 num_tags=15):
        super().__init__()

        self.input_proj = nn.Linear(input_dim, d_model)

        encoder_layer = nn.TransformerEncoderLayer(
            d_model=d_model,
            nhead=nhead,
            dim_feedforward=d_model * 4,
            dropout=0.1,
            batch_first=True
        )
        self.transformer = nn.TransformerEncoder(encoder_layer,
           num_layers)

        # Learnable query for prediction
        self.pred_query = nn.Parameter(torch.randn(1, 1, d_model))

        self.fingerprint_head = nn.Linear(d_model, input_dim)
        self.tag_head = nn.Linear(d_model, num_tags)

    def forward(self, sequence):
        # Project to model dimension
        x = self.input_proj(sequence)  # (B, L, d_model)

        # Add prediction query
        B = x.shape[0]
        query = self.pred_query.expand(B, -1, -1)
        x = torch.cat([x, query], dim=1)  # (B, L+1, d_model)

        # Transformer encoding
        encoded = self.transformer(x)

        # Use last position (the query) for prediction
        pred_hidden = encoded[:, -1, :]
```

```
        fingerprint = self.fingerprint_head(pred_hidden)
        tags = torch.sigmoid(self.tag_head(pred_hidden))

        return fingerprint, tags
```

4.4 Dataset Preparation

```
from torch.utils.data import Dataset
import json
import glob

class BlockSequenceDataset(Dataset):
    """
    Dataset of consecutive block fingerprints for prediction training.
    """
    def __init__(self, artifacts_dir, sequence_length=10):
        self.seq_len = sequence_length

        # Load all block summaries
        self.fingerprints = []
        self.tags = []

        for f in sorted(glob.glob(f"{artifacts_dir}/summaries/*.json"))
            :
            with open(f) as fp:
                data = json.load(fp)
                self.fingerprints.append(data['foldedBlockHex'])
                self.tags.append(data['tags'])

        # Convert to tensors
        self.fingerprints = torch.tensor([
            hex_to_vector(fp) for fp in self.fingerprints
        ])

        # One-hot encode tags
        self.tag_vectors = self._encode_tags(self.tags)

    def __len__(self):
        return len(self.fingerprints) - self.seq_len

    def __getitem__(self, idx):
        # Input: seq_len consecutive fingerprints
        sequence = self.fingerprints[idx:idx + self.seq_len]

        # Target: next fingerprint and tags
        target_fp = self.fingerprints[idx + self.seq_len]
        target_tags = self.tag_vectors[idx + self.seq_len]

        return sequence, target_fp, target_tags
```

5 ML Enhancement #4: Learned Semantic Tags

5.1 Current Approach

Tags are assigned via hand-coded thresholds:

```
// Current: Hard-coded thresholds
if (vector[2] > 0.55) tags.add('DEX_ACTIVITY');
if (vector[5] > 0.70) tags.add('HIGH_FEE');
if (vector[8] > 0.60) tags.add('WHALE_MOVEMENT');
```

5.2 Multi-Label Classifier

Replace with a learned classifier that captures feature correlations:

```
class TagClassifier(nn.Module):
    """
    Multi-label classifier for semantic tags.
    Input: 96-dim fingerprint
    Output: Probability for each of 15+ tags
    """
    def __init__(self, input_dim=96, num_tags=15, hidden_dims=[64, 32])
        :
        super().__init__()

        layers = []
        prev_dim = input_dim
        for h in hidden_dims:
            layers.extend([
                nn.Linear(prev_dim, h),
                nn.ReLU(),
                nn.Dropout(0.2),
                nn.BatchNorm1d(h)
            ])
            prev_dim = h

        layers.append(nn.Linear(prev_dim, num_tags))
        self.net = nn.Sequential(*layers)

    def forward(self, fingerprint):
        logits = self.net(fingerprint)
        return torch.sigmoid(logits)

    def predict_tags(self, fingerprint, threshold=0.5):
        """Return list of predicted tag names."""
        probs = self.forward(fingerprint)
        indices = torch.where(probs > threshold)[0]
        return [TAG_NAMES[i] for i in indices.tolist()]

TAG_NAMES = [
    'DEX_ACTIVITY', 'HIGH_FEE', 'WHALE_MOVEMENT', 'NFT_MINT',
    'BRIDGE_ACTIVITY', 'LENDING_ACTIVITY', 'GOVERNANCE_VOTE',
    'TOKEN_LAUNCH', 'LIQUIDATION', 'AIRDROP', 'MEV_ACTIVITY',
    'STAKING', 'MIXED_ACTIVITY', 'LOW_ACTIVITY', 'HIGH_THROUGHPUT'
]

# Training with label smoothing for robustness
def train_tag_classifier(model, train_loader, epochs=100):
    optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
    criterion = nn.BCEWithLogitsLoss(
        pos_weight=compute_class_weights(train_loader)  # Handle
            imbalance
    )
```

```
    for epoch in range(epochs):
        for fingerprints, labels in train_loader:
            optimizer.zero_grad()
            logits = model.net(fingerprints)  # Before sigmoid
            loss = criterion(logits, labels)
            loss.backward()
            optimizer.step()
```

5.3 Label Collection Strategy

To train the classifier, we need labeled data. Options:

1. **Manual labeling:** Review blocks and assign tags (expensive)
2. **Weak supervision:** Use on-chain events as labels
3. **Semi-supervised:** Use current thresholds as noisy labels, then refine

```
def generate_weak_labels(block_data):
    """
    Generate weak labels from on-chain events.
    These can be used to bootstrap the classifier.
    """
    labels = set()

    # DEX activity: Uniswap/Sushiswap router interactions
    dex_routers = {'0x7a250d...', '0xd9e1ce...'}
    if any(tx['to'] in dex_routers for tx in block_data['transactions']
        ]):
        labels.add('DEX_ACTIVITY')

    # High fee: Top 10% by gas price
    gas_prices = [tx['gasPrice'] for tx in block_data['transactions']]
    if np.mean(gas_prices) > np.percentile(historical_gas, 90):
        labels.add('HIGH_FEE')

    # NFT mint: ERC-721 Transfer with from=0x0
    for tx in block_data['transactions']:
        if is_nft_mint(tx):
            labels.add('NFT_MINT')
            break

    return labels
```

6 ML Enhancement #5: Graph Neural Network for Hypergraph

6.1 Current Hypergraph Construction

Hotzones are connected based on:

- Euclidean distance in vector space
- Product of densities

6.2 GNN-Based Learning

Learn to predict anomalous subgraphs and hotzone importance:

```
import torch
import torch.nn.functional as F
from torch_geometric.nn import GCNConv, global_mean_pool

class HotzoneGNN(nn.Module):
    """
    Graph Neural Network for hotzone analysis.
    Learns structural patterns in the hypergraph.
    """
    def __init__(self, node_dim=16, hidden_dim=32, num_layers=3):
        super().__init__()

        self.convs = nn.ModuleList()
        self.convs.append(GCNConv(node_dim, hidden_dim))
        for _ in range(num_layers - 1):
            self.convs.append(GCNConv(hidden_dim, hidden_dim))

        # Node-level anomaly prediction
        self.node_classifier = nn.Linear(hidden_dim, 1)

        # Graph-level classification (overall anomaly)
        self.graph_classifier = nn.Sequential(
            nn.Linear(hidden_dim, hidden_dim // 2),
            nn.ReLU(),
            nn.Linear(hidden_dim // 2, 1)
        )

    def forward(self, x, edge_index, batch):
        """
        Args:
            x: Node features (hotzone centroids)
            edge_index: Graph connectivity
            batch: Batch assignment for each node
        """
        # Message passing
        for conv in self.convs:
            x = F.relu(conv(x, edge_index))
            x = F.dropout(x, p=0.2, training=self.training)

        # Node-level scores
        node_scores = torch.sigmoid(self.node_classifier(x))

        # Graph-level pooling and classification
        graph_embed = global_mean_pool(x, batch)
        graph_score = torch.sigmoid(self.graph_classifier(graph_embed))

        return node_scores, graph_score

def build_graph_from_hotzones(hotzones, threshold=0.5):
    """
    Convert hotzones to PyG graph format.
    """
    from torch_geometric.data import Data

    # Node features: hotzone centroids
```

```
    x = torch.tensor([h['centroid'] for h in hotzones], dtype=torch.
        float)

    # Build edges based on distance
    edges = []
    for i, h1 in enumerate(hotzones):
        for j, h2 in enumerate(hotzones):
            if i < j:
                dist = np.linalg.norm(
                    np.array(h1['centroid']) - np.array(h2['centroid'])
                )
                if dist < threshold:
                    edges.append([i, j])
                    edges.append([j, i])  # Undirected

    edge_index = torch.tensor(edges, dtype=torch.long).t().contiguous()

    return Data(x=x, edge_index=edge_index)
```

7 Zero-Knowledge Compatibility

7.1 The Challenge

If we use learned models, how do we maintain ZK verifiability?

Core issue: The ZK proof currently verifies the deterministic pipeline (vectorization → folding → PQ). Adding ML inference introduces:

1. Floating-point operations (hard to prove in circuit)
2. Complex non-linearities (expensive to prove)
3. Large model weights (increases proof size)

7.2 Solution Options

7.2.1 Option 1: Off-Chain ML (Recommended)

On-Chain (ZK Proven)	Off-Chain (ML Enhanced)
Vectorization	Anomaly detection
Folding	Tag classification
PQ encoding	Predictions
Commitment computation	

The core fingerprint remains ZK-verifiable. ML provides additional insights but isn't proven.

7.2.2 Option 2: Quantized Models in Circuit

Convert models to fixed-point arithmetic and prove inference:

```
# Quantize model to 8-bit integers
def quantize_model(model, calibration_data):
    import torch.quantization as quant
```

```
    model.eval()
    model.qconfig = quant.get_default_qconfig('fbgemm')

    # Fuse layers
    model_fused = quant.fuse_modules(model, [['conv', 'bn', 'relu']])

    # Calibrate
    model_prepared = quant.prepare(model_fused)
    with torch.no_grad():
        for batch in calibration_data:
            model_prepared(batch)

    # Convert to quantized
    model_quantized = quant.convert(model_prepared)

    return model_quantized
```

Then implement integer-only inference in the ZK circuit. This is expensive but possible for small models.

7.2.3 Option 3: Commitment to Model Weights

Prove that a specific model was used:

$$C_{\text{model}} = \text{Hash}(\theta)$$

The verifier checks:

- The fingerprint proof is valid
- The model commitment matches a known good model
- (Optionally) The inference can be reproduced

8 Implementation Roadmap

8.1 Phase 1: Quick Wins (1-2 weeks)

1. **Export training data:** Create script to dump historical fingerprints
2. **Train PQ codebook:** Run k-means on exported data
3. **Add Isolation Forest:** Secondary anomaly signal
4. **Integrate:** Load learned codebook, show both anomaly scores

8.2 Phase 2: Prediction Model (2-4 weeks)

1. **Build sequence dataset:** Consecutive fingerprints with labels
2. **Train LSTM predictor:** Start with simple architecture
3. **Serve predictions:** Replace rule-based with model inference
4. **Add confidence intervals:** Quantify uncertainty

8.3 Phase 3: Full ML Pipeline (1-3 months)

1. **Tag classifier:** Learn from weak labels
2. **Autoencoder:** Deep anomaly detection
3. **GNN:** Hypergraph analysis
4. **Joint training:** End-to-end optimization

8.4 File Structure

```
yysfold/
|-- ml/
|   |-- __init__.py
|   |-- codebook_trainer.py      # K-means codebook training
|   |-- anomaly/
|   |   |-- isolation_forest.py  # Quick anomaly detection
|   |   |-- autoencoder.py       # Deep anomaly detection
|   |-- prediction/
|   |   |-- lstm_predictor.py    # LSTM-based predictor
|   |   |-- transformer.py       # Transformer alternative
|   |-- tags/
|   |   |-- classifier.py        # Multi-label tag classifier
|   |-- graph/
|   |   |-- gnn.py               # Hotzone GNN
|   |-- models/                  # Saved weights
|   |   |-- codebook_v1.json
|   |   |-- anomaly_v1.pt
|   |   |-- predictor_v1.pt
|-- scripts/
|   |-- exportTrainingData.ts    # Export fingerprints for Python
|   |-- trainCodebook.sh         # Run codebook training
|   |-- trainPredictor.sh        # Run predictor training
```

9 Conclusion

YYSFold's current statistical foundation provides an excellent base for ML enhancement. The recommended progression:

1. **Immediate:** Learned PQ codebook (biggest bang for buck)
2. **Short-term:** Isolation Forest anomaly detection
3. **Medium-term:** LSTM/Transformer prediction
4. **Long-term:** Full neural pipeline with GNN

The key insight is that ML enhancements can operate *alongside* the ZK-proven pipeline, providing additional signals without compromising verifiability. As quantized neural networks in ZK circuits mature, more components can be brought on-chain.