1. ResNet50+ELM-LRF Hybrid Architecture

1.1 ResNet50 Feature Extractor

```python
import torch
import torch.nn as nn
import torchvision.models as models
import numpy as np
from sklearn.preprocessing import StandardScaler

class ResNet50FeatureExtractor(nn.Module):
    """
    ResNet50 backbone for feature extraction in cooking oil quality detection
    Modified for multi-modal sensor input processing
    """
    def __init__(self, pretrained=True, freeze_backbone=False):
        super(ResNet50FeatureExtractor, self).__init__()

        # Load pretrained ResNet50
        self.backbone = models.resnet50(pretrained=pretrained)

        # Remove final classification layers
        self.features = nn.Sequential(*list(self.backbone.children())[:-2])

        # Add adaptive pooling for consistent output size
        self.adaptive_pool = nn.AdaptiveAvgPool2d((7, 7))

        # Feature dimension: 2048 * 7 * 7 = 100,352
        self.feature_dim = 100352

        if freeze_backbone:
            for param in self.features.parameters():
                param.requires_grad = False

    def forward(self, x):
        """
        Forward pass through ResNet50 feature extractor
        Args:
            x: Input tensor [batch_size, channels, height, width]
        Returns:
            features: Extracted features [batch_size, feature_dim]
        """
        features = self.features(x)
        features = self.adaptive_pool(features)
        features = features.view(features.size(0), -1)
        return features

# Multi-modal input preprocessing for sensor data
class MultiModalPreprocessor:
    """
    Preprocesses multi-modal sensor data for cooking oil quality detection
    Handles visual, chemical gas sensor, and NIR spectroscopy data
    """
    def __init__(self):
        self.visual_scaler = StandardScaler()
        self.gas_scaler = StandardScaler()
        self.nir_scaler = StandardScaler()

    def preprocess_visual(self, images):
        """Preprocess visual images"""
        # Normalize to [0, 1] and apply ImageNet normalization
        images = images.astype(np.float32) / 255.0
        mean = np.array([0.485, 0.456, 0.406])
        std = np.array([0.229, 0.224, 0.225])

        for i in range(3):
            images[:, :, :, i] = (images[:, :, :, i] - mean[i]) / std[i]

        return torch.tensor(images.transpose(0, 3, 1, 2), dtype=torch.float32)

    def preprocess_gas_sensors(self, gas_data):
        """Preprocess chemical gas sensor data"""
        # Expected features: [CO, CO2, VOCs, NH3, H2S, temperature, humidity]
        gas_normalized = self.gas_scaler.fit_transform(gas_data)
        return torch.tensor(gas_normalized, dtype=torch.float32)

    def preprocess_nir_spectroscopy(self, nir_data):
        """Preprocess Near-Infrared spectroscopy data"""
        # Wavelength range typically 700-2500 nm
        nir_normalized = self.nir_scaler.fit_transform(nir_data)
        return torch.tensor(nir_normalized, dtype=torch.float32)
```

1.2 ELM-LRF Implementation

```python
class ELM_LRF(nn.Module):
    """
    Extreme Learning Machine with Local Receptive Fields
    Based on the paper: "Local Receptive Fields based Extreme Learning Machine"
    """
    def __init__(self, input_dim, hidden_nodes, output_dim,
                 receptive_field_size=3, stride=1, C=1000):
        super(ELM_LRF, self).__init__()

        self.input_dim = input_dim
        self.hidden_nodes = hidden_nodes
        self.output_dim = output_dim
        self.receptive_field_size = receptive_field_size
        self.stride = stride
        self.C = C  # Regularization parameter
```

```python
        # Calculate number of local receptive fields
        self.num_fields = self._calculate_num_fields()

        # Initialize random weights and biases for each receptive field
        self.weights = []
        self.biases = []

        for _ in range(self.num_fields):
            # Random weights for local receptive field
            w = torch.randn(self.receptive_field_size, hidden_nodes)
            b = torch.randn(hidden_nodes)
            self.weights.append(w)
            self.biases.append(b)

        # Output weights (to be calculated during training)
        self.output_weights = None

    def _calculate_num_fields(self):
        """Calculate number of local receptive fields"""
        return (self.input_dim - self.receptive_field_size) // self.stride + 1

    def _apply_local_receptive_fields(self, x):
        """Apply local receptive fields to input"""
        batch_size = x.size(0)
        H = []

        for i, (w, b) in enumerate(zip(self.weights, self.biases)):
            # Extract local receptive field
            start_idx = i * self.stride
            end_idx = start_idx + self.receptive_field_size

            if end_idx <= self.input_dim:
                local_input = x[:, start_idx:end_idx]

                # Apply transformation: tanh(x*w + b)
                h = torch.tanh(torch.mm(local_input, w) + b.unsqueeze(0))
                H.append(h)

        # Concatenate all local features
        H = torch.cat(H, dim=1)
        return H

    def fit(self, X, Y):
        """
        Train the ELM-LRF model
        Args:
            X: Input features [batch_size, input_dim]
            Y: Target labels [batch_size, output_dim]
        """
        # Apply local receptive fields
        H = self._apply_local_receptive_fields(X)

        # Calculate output weights using Moore-Penrose pseudoinverse
        # β = (H^T H + I/C)^(-1) H^T Y
        H_T = H.t()
        if H.size(0) >= H.size(1):
            # More samples than hidden nodes
            self.output_weights = torch.mm(
                torch.mm(torch.inverse(torch.mm(H_T, H) +
                        torch.eye(H.size(1)) / self.C), H_T), Y)
        else:
            # Fewer samples than hidden nodes
            self.output_weights = torch.mm(H_T,
                torch.mm(torch.inverse(torch.mm(H, H_T) +
                        torch.eye(H.size(0)) / self.C), Y))

    def forward(self, x):
        """Forward pass through ELM-LRF"""
        H = self._apply_local_receptive_fields(x)
        output = torch.mm(H, self.output_weights)
        return output
```

1.3 Complete ResNet50+ELM-LRF Hybrid Model

```python
class ResNet50_ELM_LRF(nn.Module):
    """
    Complete hybrid model combining ResNet50 feature extraction with ELM-LRF classification
    for cooking oil reuse detection
    """
    def __init__(self, num_classes=4, hidden_nodes=1000):
        super(ResNet50_ELM_LRF, self).__init__()

        # Feature extractor
        self.feature_extractor = ResNet50FeatureExtractor(pretrained=True)

        # Dimension reduction layer
        self.feature_reducer = nn.Linear(100352, 2048)
        self.dropout = nn.Dropout(0.5)

        # ELM-LRF classifier
        self.elm_lrf = ELM_LRF(
            input_dim=2048,
            hidden_nodes=hidden_nodes,
            output_dim=num_classes,
            receptive_field_size=64,
            stride=32
        )

        self.num_classes = num_classes
        self.is_fitted = False

    def extract_features(self, x):
        """Extract features using ResNet50 backbone"""
        features = self.feature_extractor(x)
        features = self.feature_reducer(features)
        features = self.dropout(features)
        return features

    def fit(self, dataloader, device='cuda'):
        """
        Fit the ELM-LRF classifier on extracted features
        Args:
            dataloader: PyTorch DataLoader with (images, labels)
            device: Computing device
        """
```

```
        self.to(device)
        self.eval()

        all_features = []
        all_labels = []

        with torch.no_grad():
            for batch_idx, (images, labels) in enumerate(dataloader):
                images, labels = images.to(device), labels.to(device)

                # Extract features
                features = self.extract_features(images)

                all_features.append(features.cpu())
                all_labels.append(labels.cpu())

        # Concatenate all features and labels
        X = torch.cat(all_features, dim=0)
        Y = torch.cat(all_labels, dim=0)

        # Convert labels to one-hot encoding
        Y_onehot = torch.zeros(Y.size(0), self.num_classes)
        Y_onehot.scatter_(1, Y.unsqueeze(1), 1)

        # Fit ELM-LRF
        self.elm_lrf.fit(X, Y_onehot)
        self.is_fitted = True

    def forward(self, x):
        """Forward pass through complete model"""
        if not self.is_fitted:
            raise RuntimeError("Model must be fitted before inference")

        features = self.extract_features(x)
        output = self.elm_lrf(features)
        return output
```

2. VGG16+ELM-LRF Hybrid Architecture

```
class VGG16FeatureExtractor(nn.Module):
    """
    VGG16 backbone for feature extraction in cooking oil quality detection
    """
    def __init__(self, pretrained=True, freeze_backbone=False):
        super(VGG16FeatureExtractor, self).__init__()

        # Load pretrained VGG16
        vgg = models.vgg16(pretrained=pretrained)

        # Extract features (convolutional layers)
        self.features = vgg.features

        # Add adaptive pooling
        self.adaptive_pool = nn.AdaptiveAvgPool2d((7, 7))

        # Feature dimension: 512 * 7 * 7 = 25,088
        self.feature_dim = 25088

        if freeze_backbone:
            for param in self.features.parameters():
                param.requires_grad = False

    def forward(self, x):
        features = self.features(x)
        features = self.adaptive_pool(features)
        features = features.view(features.size(0), -1)
        return features

class VGG16_ELM_LRF(nn.Module):
    """
    VGG16+ELM-LRF hybrid model for cooking oil reuse detection
    """
    def __init__(self, num_classes=4, hidden_nodes=1000):
        super(VGG16_ELM_LRF, self).__init__()

        # Feature extractor
        self.feature_extractor = VGG16FeatureExtractor(pretrained=True)

        # Dimension reduction
        self.feature_reducer = nn.Linear(25088, 2048)
        self.dropout = nn.Dropout(0.5)

        # ELM-LRF classifier
        self.elm_lrf = ELM_LRF(
            input_dim=2048,
            hidden_nodes=hidden_nodes,
            output_dim=num_classes,
            receptive_field_size=64,
            stride=32
        )

        self.num_classes = num_classes
        self.is_fitted = False

    def extract_features(self, x):
        features = self.feature_extractor(x)
        features = self.feature_reducer(features)
        features = self.dropout(features)
        return features

    def fit(self, dataloader, device='cuda'):
        """Fit the model using the same procedure as ResNet50+ELM-LRF"""
        self.to(device)
        self.eval()

        all_features = []
        all_labels = []

        with torch.no_grad():
            for batch_idx, (images, labels) in enumerate(dataloader):
                images, labels = images.to(device), labels.to(device)
                features = self.extract_features(images)
                all_features.append(features.cpu())
                all_labels.append(labels.cpu())
```

```python
        X = torch.cat(all_features, dim=0)
        Y = torch.cat(all_labels, dim=0)

        # Convert to one-hot
        Y_onehot = torch.zeros(Y.size(0), self.num_classes)
        Y_onehot.scatter_(1, Y.unsqueeze(1), 1)

        self.elm_lrf.fit(X, Y_onehot)
        self.is_fitted = True

    def forward(self, x):
        if not self.is_fitted:
            raise RuntimeError("Model must be fitted before inference")

        features = self.extract_features(x)
        output = self.elm_lrf(features)
        return output
```

3. Multi-Modal Sensor Fusion

```python
class MultiModalSensorFusion(nn.Module):
    """
    Multi-modal sensor fusion for cooking oil quality detection
    Combines visual, chemical gas sensor, and NIR spectroscopy data
    """
    def __init__(self, fusion_strategy='late', num_classes=4):
        super(MultiModalSensorFusion, self).__init__()

        self.fusion_strategy = fusion_strategy
        self.num_classes = num_classes

        # Visual processing branch (ResNet50)
        self.visual_backbone = ResNet50FeatureExtractor()
        self.visual_reducer = nn.Linear(100352, 512)

        # Gas sensor processing branch
        self.gas_processor = nn.Sequential(
            nn.Linear(7, 64),  # 7 gas sensor features
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(64, 128),
            nn.ReLU(),
            nn.Linear(128, 256)
        )

        # NIR spectroscopy processing branch
        self.nir_processor = nn.Sequential(
            nn.Linear(1024, 512),  # NIR spectral features
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(512, 256),
            nn.ReLU(),
            nn.Linear(256, 256)
        )

        # Fusion and classification layers
        if fusion_strategy == 'early':
            self.fusion_dim = 512 + 256 + 256  # 1024
        elif fusion_strategy == 'late':
            self.fusion_dim = 512 + 256 + 256  # 1024

        self.fusion_layer = nn.Sequential(
            nn.Linear(self.fusion_dim, 512),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(512, 256),
            nn.ReLU(),
            nn.Linear(256, num_classes)
        )

        # Attention mechanism for modality weighting
        self.attention = nn.Sequential(
            nn.Linear(self.fusion_dim, 128),
            nn.ReLU(),
            nn.Linear(128, 3),  # 3 modalities
            nn.Softmax(dim=1)
        )

    def forward(self, visual_data, gas_data, nir_data):
        """
        Forward pass through multi-modal fusion network
        Args:
            visual_data: Visual images [batch_size, 3, H, W]
            gas_data: Gas sensor readings [batch_size, 7]
            nir_data: NIR spectroscopy data [batch_size, 1024]
        """
        # Process each modality
        visual_features = self.visual_backbone(visual_data)
        visual_features = self.visual_reducer(visual_features)

        gas_features = self.gas_processor(gas_data)
        nir_features = self.nir_processor(nir_data)

        # Fusion strategy
        if self.fusion_strategy == 'early':
            # Concatenate all features
            fused_features = torch.cat([visual_features, gas_features, nir_features], dim=1)

        elif self.fusion_strategy == 'late':
            # Weighted fusion using attention
            all_features = torch.cat([visual_features, gas_features, nir_features], dim=1)
            attention_weights = self.attention(all_features)

            # Apply attention weights
            weighted_visual = visual_features * attention_weights[:, 0].unsqueeze(1)
            weighted_gas = gas_features * attention_weights[:, 1].unsqueeze(1)
            weighted_nir = nir_features * attention_weights[:, 2].unsqueeze(1)

            fused_features = torch.cat([weighted_visual, weighted_gas, weighted_nir], dim=1)

        # Final classification
        output = self.fusion_layer(fused_features)
        return output, attention_weights if self.fusion_strategy == 'late' else None
```

4. Data Preprocessing and Augmentation

```python
import cv2
import albumentations as A
from albumentations.pytorch import ToTensorV2

class CookingOilDataPreprocessor:
    """
    Comprehensive data preprocessing for cooking oil quality detection
    """
    def __init__(self, image_size=(224, 224), augment=True):
        self.image_size = image_size
        self.augment = augment

        # Image preprocessing pipeline
        self.train_transform = A.Compose([
            A.Resize(height=image_size[0], width=image_size[1]),
            A.HorizontalFlip(p=0.5),
            A.VerticalFlip(p=0.3),
            A.RandomRotate90(p=0.3),
            A.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1, p=0.5),
            A.GaussianBlur(blur_limit=(3, 7), p=0.3),
            A.GaussNoise(var_limit=(10.0, 50.0), p=0.3),
            A.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
            ToTensorV2()
        ])

        self.val_transform = A.Compose([
            A.Resize(height=image_size[0], width=image_size[1]),
            A.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
            ToTensorV2()
        ])

    def preprocess_image(self, image, training=True):
        """Preprocess cooking oil images"""
        if isinstance(image, str):
            image = cv2.imread(image)
            image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

        if training and self.augment:
            transformed = self.train_transform(image=image)
        else:
            transformed = self.val_transform(image=image)

        return transformed['image']

    def normalize_sensor_data(self, sensor_data, sensor_type='gas'):
        """
        Normalize sensor data based on type
        Args:
            sensor_data: Raw sensor readings
            sensor_type: 'gas', 'nir', or 'temperature'
        """
        if sensor_type == 'gas':
            # Gas sensor normalization (CO, CO2, VOCs, NH3, H2S, temp, humidity)
            gas_ranges = {
                'CO': (0, 1000),      # ppm
                'CO2': (300, 5000),   # ppm
                'VOCs': (0, 500),     # ppb
                'NH3': (0, 100),      # ppm
                'H2S': (0, 50),       # ppm
                'temp': (15, 45),     # Celsius
                'humidity': (20, 80)  # %
            }

            normalized_data = np.zeros_like(sensor_data)
            for i, (key, (min_val, max_val)) in enumerate(gas_ranges.items()):
                normalized_data[:, i] = (sensor_data[:, i] - min_val) / (max_val - min_val)
                normalized_data[:, i] = np.clip(normalized_data[:, i], 0, 1)

            return normalized_data

        elif sensor_type == 'nir':
            # NIR spectroscopy normalization
            # Apply Standard Normal Variate (SNV) correction
            mean_spectra = np.mean(sensor_data, axis=1, keepdims=True)
            std_spectra = np.std(sensor_data, axis=1, keepdims=True)
            snv_data = (sensor_data - mean_spectra) / (std_spectra + 1e-8)

            # Apply Savitzky-Golay smoothing
            from scipy.signal import savgol_filter
            smoothed_data = savgol_filter(snv_data, window_length=15, polyorder=2, axis=1)

            return smoothed_data

    def extract_chemical_features(self, oil_samples):
        """
        Extract chemical features from oil samples
        Simulates laboratory analysis results
        """
        features = {
            'polar_compounds': [],
            'free_fatty_acids': [],
            'peroxide_value': [],
            'pahs_concentration': [],
            'aldehydes_concentration': [],
            'oxidation_index': []
        }

        for sample in oil_samples:
            reuse_cycles = sample['reuse_cycles']

            # Simulate chemical degradation based on reuse cycles
            base_polar = 8.4 + (4.2 * reuse_cycles) + np.random.normal(0, 1.5)
            base_ffa = 0.15 + (0.08 * reuse_cycles) + np.random.normal(0, 0.03)
            base_pv = 2.1 + (3.2 * reuse_cycles) + np.random.normal(0, 0.8)

            # PAHs and aldehydes increase exponentially
            pahs = 2.3 * np.exp(0.85 * reuse_cycles) + np.random.normal(0, 3.0)
            aldehydes = 15.2 * np.exp(0.78 * reuse_cycles) + np.random.normal(0, 8.0)

            oxidation = base_polar + base_ffa * 10 + base_pv * 2
```

```
            features['polar_compounds'].append(np.clip(base_polar, 8, 50))
            features['free_fatty_acids'].append(np.clip(base_ffa, 0.1, 2.0))
            features['peroxide_value'].append(np.clip(base_pv, 1, 30))
            features['pahs_concentration'].append(np.clip(pahs, 1, 200))
            features['aldehydes_concentration'].append(np.clip(aldehydes, 10, 500))
            features['oxidation_index'].append(oxidation)

        return features
```

5. Training Protocol and Optimization

```
class CookingOilTrainer:
    """
    Training protocol for cooking oil quality detection models
    """
    def __init__(self, model, device='cuda', learning_rate=0.001):
        self.model = model
        self.device = device
        self.learning_rate = learning_rate

        # Loss function and optimizer
        self.criterion = nn.CrossEntropyLoss()
        self.optimizer = torch.optim.Adam(
            model.parameters(),
            lr=learning_rate,
            weight_decay=1e-4
        )

        # Learning rate scheduler
        self.scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
            self.optimizer,
            mode='min',
            factor=0.5,
            patience=5,
            verbose=True
        )

        # Training history
        self.train_losses = []
        self.val_losses = []
        self.train_accuracies = []
        self.val_accuracies = []

    def train_epoch(self, dataloader):
        """Train for one epoch"""
        self.model.train()
        total_loss = 0
        correct = 0
        total = 0

        for batch_idx, (visual_data, gas_data, nir_data, labels) in enumerate(dataloader):
            visual_data = visual_data.to(self.device)
            gas_data = gas_data.to(self.device)
            nir_data = nir_data.to(self.device)
            labels = labels.to(self.device)

            self.optimizer.zero_grad()

            # Forward pass
            outputs, attention_weights = self.model(visual_data, gas_data, nir_data)
            loss = self.criterion(outputs, labels)

            # Backward pass
            loss.backward()
            self.optimizer.step()

            total_loss += loss.item()
            _, predicted = outputs.max(1)
            total += labels.size(0)
            correct += predicted.eq(labels).sum().item()

        avg_loss = total_loss / len(dataloader)
        accuracy = 100.0 * correct / total

        return avg_loss, accuracy

    def validate(self, dataloader):
        """Validate the model"""
        self.model.eval()
        total_loss = 0
        correct = 0
        total = 0

        with torch.no_grad():
            for visual_data, gas_data, nir_data, labels in dataloader:
                visual_data = visual_data.to(self.device)
                gas_data = gas_data.to(self.device)
                nir_data = nir_data.to(self.device)
                labels = labels.to(self.device)

                outputs, _ = self.model(visual_data, gas_data, nir_data)
                loss = self.criterion(outputs, labels)

                total_loss += loss.item()
                _, predicted = outputs.max(1)
                total += labels.size(0)
                correct += predicted.eq(labels).sum().item()

        avg_loss = total_loss / len(dataloader)
        accuracy = 100.0 * correct / total

        return avg_loss, accuracy

    def train(self, train_loader, val_loader, epochs=50):
        """Complete training loop"""
        best_val_acc = 0

        for epoch in range(epochs):
            print(f'\nEpoch {epoch+1}/{epochs}')
            print('-' * 50)

            # Training
            train_loss, train_acc = self.train_epoch(train_loader)
            self.train_losses.append(train_loss)
            self.train_accuracies.append(train_acc)
```

```
        # Validation
        val_loss, val_acc = self.validate(val_loader)
        self.val_losses.append(val_loss)
        self.val_accuracies.append(val_acc)

        print(f'Train Loss: {train_loss:.4f}, Train Acc: {train_acc:.2f}%')
        print(f'Val Loss: {val_loss:.4f}, Val Acc: {val_acc:.2f}%')

        # Learning rate scheduling
        self.scheduler.step(val_loss)

        # Save best model
        if val_acc > best_val_acc:
            best_val_acc = val_acc
            torch.save(self.model.state_dict(), 'best_model.pth')
            print(f'New best model saved with validation accuracy: {val_acc:.2f}%')
```

6. Performance Evaluation Metrics

```
from sklearn.metrics import classification_report, confusion_matrix, roc_auc_score
from sklearn.metrics import precision_recall_fscore_support
import matplotlib.pyplot as plt
import seaborn as sns

class PerformanceEvaluator:
    """
    Comprehensive performance evaluation for cooking oil quality detection
    """
    def __init__(self, class_names=['Fresh', 'Low Reuse', 'Medium Reuse', 'High Reuse']):
        self.class_names = class_names
        self.num_classes = len(class_names)

    def evaluate_model(self, model, test_loader, device='cuda'):
        """
        Comprehensive model evaluation
        Returns metrics including accuracy, precision, recall, F1-score, and AUC
        """
        model.eval()
        all_predictions = []
        all_labels = []
        all_probabilities = []
        processing_times = []

        with torch.no_grad():
            for visual_data, gas_data, nir_data, labels in test_loader:
                visual_data = visual_data.to(device)
                gas_data = gas_data.to(device)
                nir_data = nir_data.to(device)

                # Measure processing time
                start_time = time.time()
                outputs, _ = model(visual_data, gas_data, nir_data)
                end_time = time.time()

                processing_times.append((end_time - start_time) / visual_data.size(0))

                # Get predictions and probabilities
                probabilities = torch.softmax(outputs, dim=1)
                _, predictions = outputs.max(1)

                all_predictions.extend(predictions.cpu().numpy())
                all_labels.extend(labels.numpy())
                all_probabilities.extend(probabilities.cpu().numpy())

        # Convert to numpy arrays
        y_true = np.array(all_labels)
        y_pred = np.array(all_predictions)
        y_prob = np.array(all_probabilities)

        # Calculate metrics
        accuracy = np.mean(y_true == y_pred) * 100
        precision, recall, f1, _ = precision_recall_fscore_support(
            y_true, y_pred, average='weighted'
        )

        # Calculate AUC for multi-class
        auc_scores = []
        for i in range(self.num_classes):
            y_true_binary = (y_true == i).astype(int)
            y_prob_binary = y_prob[:, i]
            auc = roc_auc_score(y_true_binary, y_prob_binary)
            auc_scores.append(auc)

        avg_auc = np.mean(auc_scores)
        avg_processing_time = np.mean(processing_times)

        # Create results dictionary
        results = {
            'accuracy': accuracy,
            'precision': precision * 100,
            'recall': recall * 100,
            'f1_score': f1 * 100,
            'auc': avg_auc,
            'processing_time': avg_processing_time,
            'confusion_matrix': confusion_matrix(y_true, y_pred),
            'classification_report': classification_report(y_true, y_pred,
                                                           target_names=self.class_names)
        }

        return results

    def plot_confusion_matrix(self, cm, title='Confusion Matrix'):
        """Plot confusion matrix"""
        plt.figure(figsize=(8, 6))
        sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
                    xticklabels=self.class_names,
                    yticklabels=self.class_names)
        plt.title(title)
        plt.ylabel('True Label')
        plt.xlabel('Predicted Label')
        plt.tight_layout()
        plt.show()
```

```python
    def compare_models(self, results_dict):
        """Compare multiple model results"""
        models = list(results_dict.keys())
        metrics = ['accuracy', 'precision', 'recall', 'f1_score']

        # Create comparison DataFrame
        comparison_data = {}
        for metric in metrics:
            comparison_data[metric] = [results_dict[model][metric] for model in models]

        comparison_df = pd.DataFrame(comparison_data, index=models)

        # Plot comparison
        fig, axes = plt.subplots(2, 2, figsize=(15, 10))
        axes = axes.flatten()

        for i, metric in enumerate(metrics):
            comparison_df[metric].plot(kind='bar', ax=axes[i], color=['#2E86AB', '#A23B72', '#F18F01', '#C73E1D'])
            axes[i].set_title(f{metric.capitalize()} Comparison')
            axes[i].set_ylabel(f{metric.capitalize()} (%)')
            axes[i].tick_params(axis='x', rotation=45)
            axes[i].grid(axis='y', alpha=0.3)

        plt.tight_layout()
        plt.show()

        return comparison_df
```

7. Chemical Analysis Correlation

```python
class ChemicalAnalysisCorrelator:
    """
    Correlates AI predictions with chemical analysis results
    """
    def __init__(self):
        # Safety thresholds based on international standards
        self.safety_thresholds = {
            'polar_compounds': 25.0,     # % max for food use
            'free_fatty_acids': 0.5,    # % max
            'peroxide_value': 10.0,     # meq O2/kg max
            'pahs_total': 50.0,         # µg/kg max
            'aldehydes_total': 100.0    # µg/kg max
        }

        # Reuse category mappings
        self.category_mapping = {
            0: 'Fresh (0 cycles)',
            1: 'Low (2-3 cycles)',
            2: 'Medium (4-6 cycles)',
            3: 'High (7+ cycles)'
        }

    def predict_chemical_parameters(self, ai_predictions, confidence_scores):
        """
        Predict chemical parameters based on AI classification results
        """
        predicted_chemicals = []

        for pred, conf in zip(ai_predictions, confidence_scores):
            # Base values for each category
            if pred == 0:  # Fresh
                chemicals = {
                    'polar_compounds': 8.4 + np.random.normal(0, 1.2),
                    'free_fatty_acids': 0.15 + np.random.normal(0, 0.03),
                    'peroxide_value': 2.1 + np.random.normal(0, 0.5),
                    'pahs_total': 2.3 + np.random.normal(0, 0.5),
                    'aldehydes_total': 15.2 + np.random.normal(0, 3.1)
                }
            elif pred == 1:  # Low reuse
                chemicals = {
                    'polar_compounds': 17.6 + np.random.normal(0, 2.8),
                    'free_fatty_acids': 0.31 + np.random.normal(0, 0.06),
                    'peroxide_value': 5.8 + np.random.normal(0, 1.2),
                    'pahs_total': 18.7 + np.random.normal(0, 4.2),
                    'aldehydes_total': 78.3 + np.random.normal(0, 12.5)
                }
            elif pred == 2:  # Medium reuse
                chemicals = {
                    'polar_compounds': 28.3 + np.random.normal(0, 4.2),
                    'free_fatty_acids': 0.54 + np.random.normal(0, 0.12),
                    'peroxide_value': 12.4 + np.random.normal(0, 2.8),
                    'pahs_total': 67.4 + np.random.normal(0, 11.8),
                    'aldehydes_total': 194.7 + np.random.normal(0, 28.9)
                }
            else:  # High reuse
                chemicals = {
                    'polar_compounds': 41.7 + np.random.normal(0, 6.8),
                    'free_fatty_acids': 0.89 + np.random.normal(0, 0.18),
                    'peroxide_value': 22.6 + np.random.normal(0, 4.5),
                    'pahs_total': 156.9 + np.random.normal(0, 24.3),
                    'aldehydes_total': 387.2 + np.random.normal(0, 45.7)
                }

            # Adjust based on confidence
            for key in chemicals:
                # Lower confidence leads to more uncertainty
                uncertainty_factor = 1.0 / conf
                chemicals[key] *= uncertainty_factor
                chemicals[key] = max(0, chemicals[key])  # Ensure non-negative

            predicted_chemicals.append(chemicals)

        return predicted_chemicals

    def assess_safety_risk(self, chemical_params):
        """
        Assess safety risk based on chemical parameters
        """
        risk_levels = []

        for params in chemical_params:
            violations = []
```

```
        for param, value in params.items():
            if param in self.safety_thresholds:
                threshold = self.safety_thresholds[param]
                if value > threshold:
                    violations.append({
                        'parameter': param,
                        'value': value,
                        'threshold': threshold,
                        'excess_ratio': value / threshold
                    })

        # Determine risk level
        if len(violations) == 0:
            risk_level = 'Safe'
        elif len(violations) <= 2 and all(v['excess_ratio'] < 2.0 for v in violations):
            risk_level = 'Moderate Risk'
        else:
            risk_level = 'High Risk'

        risk_levels.append({
            'risk_level': risk_level,
            'violations': violations,
            'total_violations': len(violations)
        })

    return risk_levels

def generate_report(self, ai_predictions, chemical_params, risk_assessments):
    """Generate comprehensive analysis report"""
    report = []

    for i, (pred, chem, risk) in enumerate(zip(ai_predictions, chemical_params, risk_assessments)):
        sample_report = {
            'sample_id': i + 1,
            'ai_prediction': self.category_mapping[pred],
            'chemical_analysis': chem,
            'safety_assessment': risk,
            'recommendation': self._get_recommendation(risk['risk_level'])
        }
        report.append(sample_report)

    return report

def _get_recommendation(self, risk_level):
    """Get recommendation based on risk level"""
    recommendations = {
        'Safe': 'Oil is safe for continued use. Monitor regularly.',
        'Moderate Risk': 'Oil quality is degraded. Consider replacement soon. Avoid high-temperature cooking.',
        'High Risk': 'IMMEDIATE REPLACEMENT REQUIRED. Oil poses health risks and should not be used for food preparation.'
    }
    return recommendations.get(risk_level, 'Assessment required')
```

8. Complete Implementation Example

```
def main():
    """
    Complete implementation example for cooking oil quality detection
    """
    # Set random seeds for reproducibility
    torch.manual_seed(42)
    np.random.seed(42)

    # Device configuration
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    print(f'Using device: {device}')

    # Initialize models
    print('Initializing models...')
    resnet_elm = ResNet50_ELM_LRF(num_classes=4, hidden_nodes=1000).to(device)
    vgg_elm = VGG16_ELM_LRF(num_classes=4, hidden_nodes=1000).to(device)
    multimodal_model = MultiModalSensorFusion(fusion_strategy='late', num_classes=4).to(device)

    # Initialize data preprocessor
    preprocessor = CookingOilDataPreprocessor(image_size=(224, 224))

    # Initialize evaluator
    evaluator = PerformanceEvaluator()
```

```python
    # Initialize chemical correlator
    chemical_correlator = ChemicalAnalysisCorrelator()

    print('Setup complete!')
    print('Models ready for training and evaluation.')

    # Example usage for inference
    print('\nExample inference process:')
    print('1. Load and preprocess multi-modal data')
    print('2. Run AI prediction')
    print('3. Correlate with chemical analysis')
    print('4. Generate safety assessment')
    print('5. Provide recommendations')

if __name__ == '__main__':
    main()
```