```
"""
SOFIS: Self-Organizing Fuzzy Inference System for REBA Assessment
Production Implementation

A complete, Demo implementation of the SOFIS system for automated Ergonomic
risk evaluation (using REBA; Rapid Entire Body Assessment) with uncertainty
quantification,
adaptive learning, and task specialization.

Author: Roy Lan; roylan@my.utsa.edu
Version: 1.0.0
License: Academic Use, Please email Author before use
"""

import os
import logging
import pickle
import json
import time
import warnings
from pathlib import Path
from typing import Dict, List, Tuple, Optional, Any, Union
from dataclasses import dataclass, asdict, field
from collections import defaultdict
from abc import ABC, abstractmethod

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from scipy import stats
from scipy.spatial.distance import euclidean, cosine
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.metrics import mean_absolute_error, mean_squared_error,
silhouette_score
from sklearn.model_selection import cross_val_score

# Suppress warnings for clean output
warnings.filterwarnings('ignore', category=UserWarning)
warnings.filterwarnings('ignore', category=FutureWarning)

# Try importing scikit-fuzzy with graceful fallback
try:
    import skfuzzy as fuzz
    from skfuzzy import control as ctrl
    FUZZY_AVAILABLE = True
except ImportError:
    FUZZY_AVAILABLE = False
```

```
    ctrl = None
    fuzz = None

# Configure logging
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(name)s -
%(levelname)s - %(message)s')

# Global constants
RANDOM_SEED = 42
np.random.seed(RANDOM_SEED)

# Set up logger
logger = logging.getLogger(__name__)


@dataclass
class SOFISConfiguration:
    """Configuration parameters for SOFIS system."""
    max_rules: int = 200
    dropout_rate: float = 0.3
    mc_samples: int = 10
    learning_rate: float = 0.05
    uncertainty_alpha: float = 4.0
    min_rule_activation: int = 3
    rule_pruning_threshold: float = 0.3
    convergence_threshold: float = 0.01
    max_epochs: int = 100
    n_clusters: int = 3
    random_seed: int = 42
    min_rules: int = 10
    silhouette_threshold: float = 0.6
    confidence_threshold: float = 0.7


@dataclass
class AssessmentResult:
    """Container for SOFIS assessment results."""
    risk_score: float
    risk_category: str
    confidence_score: float
    uncertainty_interval: Tuple[float, float]
    aleatoric_uncertainty: float
    epistemic_uncertainty: float
    total_uncertainty: float
    rule_activations: Dict[int, float]
    input_features: Dict[str, float]
    task_cluster: Optional[int] = None
    explanation: Optional[str] = None
    timestamp: float = field(default_factory=time.time)
    is_fallback: bool = False
```

```
    processing_time: float = 0.0


class FuzzySystemError(Exception):
    """Custom exception for fuzzy system errors."""
    pass


class DataValidationError(Exception):
    """Custom exception for data validation errors."""
    pass


class PoseProcessor:
    """Processes 3D pose keypoints and calculates joint angles."""

    def __init__(self):
        self.logger = logging.getLogger(self.__class__.__name__)

    def calculate_joint_angles(self, pose_keypoints: Dict[str, np.ndarray]) ->
Dict[str, float]:
        """
        Calculate 19 joint angles from 3D pose keypoints.(THIS CAN BE MODIFIED FOR
USE CASE)

        Args:
            pose_keypoints: Dictionary of 3D keypoint positions

        Returns:
            Dictionary of calculated joint angles in degrees
        """
        try:
            joint_angles = {}

            # Validate input keypoints
            required_keypoints = [
                'head', 'neck', 'left_shoulder', 'right_shoulder',
                'left_elbow', 'right_elbow', 'left_wrist', 'right_wrist',
                'left_hip', 'right_hip', 'left_knee', 'right_knee',
                'left_ankle', 'right_ankle'
            ]

            # Check for missing keypoints and provide defaults
            validated_keypoints = self._validate_and_fill_keypoints(pose_keypoints,
required_keypoints)

            # Calculate center points
            shoulder_center = (validated_keypoints['left_shoulder'] +
validated_keypoints['right_shoulder']) / 2
            hip_center = (validated_keypoints['left_hip'] +
```

```
validated_keypoints['right_hip']) / 2

            # Trunk flexion angle
            v_torso = shoulder_center - hip_center
            v_vertical = np.array([0, 1, 0])
            joint_angles['trunk_angle'] = self._angle_between_vectors(v_torso,
v_vertical)

            # Neck flexion angle
            v_head = validated_keypoints['head'] - validated_keypoints['neck']
            v_shoulder = validated_keypoints['neck'] - shoulder_center
            joint_angles['neck_angle'] = self._angle_between_vectors(v_head,
v_shoulder)

            # Shoulder and arm angles
            for side in ['left', 'right']:
                shoulder_key = f'{side}_shoulder'
                elbow_key = f'{side}_elbow'
                wrist_key = f'{side}_wrist'

                # Shoulder flexion
                v_arm = validated_keypoints[elbow_key] -
validated_keypoints[shoulder_key]
                shoulder_flex = self._angle_between_vectors(v_arm, v_torso)
                joint_angles[f'{side}_shoulder_flex_angle'] = shoulder_flex

                # Shoulder abduction
                v_shoulder_line = validated_keypoints['right_shoulder'] -
validated_keypoints['left_shoulder']
                shoulder_abd = self._angle_between_vectors(v_arm, v_shoulder_line)
                joint_angles[f'{side}_shoulder_abd_angle'] = shoulder_abd

                # Elbow flexion
                v_upper_arm = validated_keypoints[elbow_key] -
validated_keypoints[shoulder_key]
                v_forearm = validated_keypoints[wrist_key] -
validated_keypoints[elbow_key]
                elbow_angle = 180 - self._angle_between_vectors(v_upper_arm,
v_forearm)
                joint_angles[f'{side}_elbow_angle'] = elbow_angle

                # Wrist angles (simplified)
                wrist_flex = self._calculate_wrist_flexion(v_forearm)
                joint_angles[f'{side}_wrist_flex_angle'] = wrist_flex
                joint_angles[f'{side}_wrist_dev_angle'] = abs(wrist_flex) * 0.7  #
Approximation

            # Leg angles
            for side in ['left', 'right']:
                hip_key = f'{side}_hip'
```

```
                knee_key = f'{side}_knee'
                ankle_key = f'{side}_ankle'

                # Hip flexion
                v_thigh = validated_keypoints[knee_key] - 
validated_keypoints[hip_key]
                hip_angle = self._angle_between_vectors(v_thigh, v_vertical)
                joint_angles[f'{side}_hip_angle'] = hip_angle

                # Knee flexion
                v_shin = validated_keypoints[ankle_key] - 
validated_keypoints[knee_key]
                knee_angle = 180 - self._angle_between_vectors(v_thigh, v_shin)
                joint_angles[f'{side}_knee_angle'] = knee_angle

                # Ankle angle (simplified)
                ankle_angle = self._calculate_ankle_angle(v_shin)
                joint_angles[f'{side}_ankle_angle'] = ankle_angle

            # Calculate representative angles
            joint_angles['upper_arm_angle'] = max(
                joint_angles['left_shoulder_flex_angle'],
                joint_angles['right_shoulder_flex_angle']
            )
            joint_angles['lower_arm_angle'] = np.mean([
                joint_angles['left_elbow_angle'],
                joint_angles['right_elbow_angle']
            ])
            joint_angles['wrist_flex_angle'] = np.mean([
                joint_angles['left_wrist_flex_angle'],
                joint_angles['right_wrist_flex_angle']
            ])
            joint_angles['wrist_dev_angle'] = np.mean([
                joint_angles['left_wrist_dev_angle'],
                joint_angles['right_wrist_dev_angle']
            ])
            joint_angles['hip_angle'] = np.mean([
                joint_angles['left_hip_angle'],
                joint_angles['right_hip_angle']
            ])
            joint_angles['knee_angle'] = np.mean([
                joint_angles['left_knee_angle'],
                joint_angles['right_knee_angle']
            ])
            joint_angles['ankle_angle'] = np.mean([
                joint_angles['left_ankle_angle'],
                joint_angles['right_ankle_angle']
            ])
            joint_angles['leg_angle'] = joint_angles['knee_angle']
```

```
            # Additional derived angles
            joint_angles['spine_angle'] = abs(joint_angles['trunk_angle'] - 90)
            joint_angles['head_angle'] = joint_angles['neck_angle']

            # Ensure all angles are in valid ranges
            for angle_name in joint_angles:
                joint_angles[angle_name] = np.clip(joint_angles[angle_name], 0, 180)

            self.logger.debug(f"Calculated {len(joint_angles)} joint angles")
            return joint_angles

        except Exception as e:
            self.logger.error(f"Error calculating joint angles: {e}")
            raise PoseProcessor.ProcessingError(f"Failed to calculate joint angles:
{e}")

    def _validate_and_fill_keypoints(self, keypoints: Dict[str, np.ndarray],
                                     required: List[str]) -> Dict[str, np.ndarray]:
        """Validate keypoints and fill missing ones with reasonable defaults."""
        validated = {}

        # Default pose (neutral standing position)
        default_positions = {
            'head': np.array([0, 1.7, 0]),
            'neck': np.array([0, 1.6, 0]),
            'left_shoulder': np.array([-0.2, 1.5, 0]),
            'right_shoulder': np.array([0.2, 1.5, 0]),
            'left_elbow': np.array([-0.3, 1.2, 0]),
            'right_elbow': np.array([0.3, 1.2, 0]),
            'left_wrist': np.array([-0.35, 0.9, 0]),
            'right_wrist': np.array([0.35, 0.9, 0]),
            'left_hip': np.array([-0.1, 1.0, 0]),
            'right_hip': np.array([0.1, 1.0, 0]),
            'left_knee': np.array([-0.1, 0.5, 0]),
            'right_knee': np.array([0.1, 0.5, 0]),
            'left_ankle': np.array([-0.1, 0.0, 0]),
            'right_ankle': np.array([0.1, 0.0, 0])
        }

        for keypoint in required:
            if keypoint in keypoints and keypoints[keypoint] is not None:
                validated[keypoint] = np.array(keypoints[keypoint])
            else:
                validated[keypoint] = default_positions.get(keypoint, np.array([0,
0, 0]))
                self.logger.warning(f"Using default position for missing keypoint:
{keypoint}")

        return validated
```

```python
    def _angle_between_vectors(self, v1: np.ndarray, v2: np.ndarray) -> float:
        """Calculate angle between two 3D vectors in degrees."""
        v1_norm = v1 / (np.linalg.norm(v1) + 1e-8)
        v2_norm = v2 / (np.linalg.norm(v2) + 1e-8)
        cos_angle = np.clip(np.dot(v1_norm, v2_norm), -1.0, 1.0)
        angle_rad = np.arccos(cos_angle)
        return np.degrees(angle_rad)

    def _calculate_wrist_flexion(self, forearm_vector: np.ndarray) -> float:
        """Calculate wrist flexion angle from forearm vector."""
        # Simplified calculation based on forearm orientation
        horizontal = np.array([1, 0, 0])
        angle = self._angle_between_vectors(forearm_vector, horizontal)
        return min(angle, 90)  # Limit to reasonable range

    def _calculate_ankle_angle(self, shin_vector: np.ndarray) -> float:
        """Calculate ankle angle from shin vector."""
        # Simplified calculation
        vertical = np.array([0, -1, 0])
        angle = self._angle_between_vectors(shin_vector, vertical)
        return min(angle, 90)  # Limit to reasonable range

    class ProcessingError(Exception):
        """Exception raised for pose processing errors."""
        pass


class MembershipFunctionManager:
    """Manages adaptive Gaussian membership functions for fuzzy variables."""

    def __init__(self, config: SOFISConfiguration):
        self.config = config
        self.logger = logging.getLogger(self.__class__.__name__)
        self.adaptation_history = defaultdict(list)

    def learn_membership_functions(self, data: np.ndarray, variable_name: str) ->
Dict[str, np.ndarray]:
        """
        Learn Gaussian membership functions from data using K-means clustering.

        Args:
            data: Training data for the variable
            variable_name: Name of the variable

        Returns:
            Dictionary containing membership function parameters
        """
        try:
            if len(data) < 10:
                self.logger.warning(f"Insufficient data for {variable_name}, using
```

```
defaults")
                return self._create_default_functions(data.min(), data.max())

            # Apply K-means clustering
            kmeans = KMeans(n_clusters=3, random_state=self.config.random_seed,
n_init=10)
            data_reshaped = data.reshape(-1, 1)
            cluster_labels = kmeans.fit_predict(data_reshaped)
            centers = np.sort(kmeans.cluster_centers_.flatten())

            # Calculate standard deviations for each cluster
            std_devs = []
            for i, center in enumerate(centers):
                cluster_data = data[cluster_labels == i]
                if len(cluster_data) > 1:
                    std_dev = np.std(cluster_data)
                else:
                    std_dev = (data.max() - data.min()) / 12

                # Ensure minimum spread
                min_std = (data.max() - data.min()) / 20
                std_devs.append(max(std_dev, min_std))

            # Store adaptation history
            self.adaptation_history[variable_name].append({
                'timestamp': time.time(),
                'centers': centers.tolist(),
                'std_devs': std_devs,
                'data_size': len(data)
            })

            mf_params = {
                'low': {'center': centers[0], 'sigma': std_devs[0]},
                'medium': {'center': centers[1], 'sigma': std_devs[1]},
                'high': {'center': centers[2], 'sigma': std_devs[2]}
            }

            self.logger.debug(f"Learned membership functions for {variable_name}")
            return mf_params

        except Exception as e:
            self.logger.error(f"Error learning membership functions for
{variable_name}: {e}")
            return self._create_default_functions(data.min(), data.max())

    def _create_default_functions(self, min_val: float, max_val: float) -> Dict[str,
Dict[str, float]]:
        """Create default evenly distributed Gaussian membership functions."""
        range_val = max_val - min_val
        centers = [min_val + 0.25*range_val, min_val + 0.5*range_val, min_val +
```

```
0.75*range_val]
        sigma = range_val / 8

        return {
            'low': {'center': centers[0], 'sigma': sigma},
            'medium': {'center': centers[1], 'sigma': sigma},
            'high': {'center': centers[2], 'sigma': sigma}
        }

    def get_membership_value(self, value: float, mf_params: Dict[str, float]) ->
float:
        """Calculate membership value for Gaussian function."""
        center = mf_params['center']
        sigma = mf_params['sigma']
        return np.exp(-0.5 * ((value - center) / sigma) ** 2)


class RuleBaseManager:
    """Manages the fuzzy rule base with dynamic evolution capabilities."""

    def __init__(self, config: SOFISConfiguration):
        self.config = config
        self.logger = logging.getLogger(self.__class__.__name__)
        self.rules = []
        self.rule_weights = {}
        self.rule_performance = {}
        self.rule_activations = defaultdict(int)
        self.rule_generation_history = []

    def initialize_systematic_rules(self) -> List[Dict[str, Any]]:
        """Initialize systematic rule base based on REBA methodology."""
        self.logger.info("Initializing systematic REBA rule base...")

        # Define systematic rule patterns
        rule_patterns = [
            # Low risk patterns
            {'conditions': [('trunk_angle', 'low'), ('neck_angle', 'low'),
('upper_arm_angle', 'low'), ('load', 'low')], 'output': 'negligible'},
            {'conditions': [('trunk_angle', 'low'), ('neck_angle', 'medium'),
('upper_arm_angle', 'low')], 'output': 'low'},
            {'conditions': [('trunk_angle', 'medium'), ('neck_angle', 'low'),
('load', 'low')], 'output': 'low'},

            # Medium risk patterns
            {'conditions': [('trunk_angle', 'medium'), ('upper_arm_angle',
'medium'), ('load', 'medium')], 'output': 'medium'},
            {'conditions': [('trunk_angle', 'low'), ('upper_arm_angle', 'high'),
('activity', 'medium')], 'output': 'medium'},
            {'conditions': [('neck_angle', 'high'), ('upper_arm_angle', 'medium')],
'output': 'medium'},
```

```
            {'conditions': [('wrist_flex_angle', 'high'), ('coupling', 'medium')],
'output': 'medium'},

            # High risk patterns
            {'conditions': [('trunk_angle', 'high'), ('neck_angle', 'medium')],
'output': 'high'},
            {'conditions': [('trunk_angle', 'medium'), ('upper_arm_angle', 'high'),
('load', 'high')], 'output': 'high'},
            {'conditions': [('upper_arm_angle', 'high'), ('activity', 'high')],
'output': 'high'},
            {'conditions': [('load', 'high'), ('coupling', 'high')], 'output':
'high'},

            # Very high risk patterns
            {'conditions': [('trunk_angle', 'high'), ('upper_arm_angle', 'high')],
'output': 'very_high'},
            {'conditions': [('trunk_angle', 'high'), ('load', 'high'), ('activity',
'high')], 'output': 'very_high'},
            {'conditions': [('neck_angle', 'high'), ('trunk_angle', 'high'),
('coupling', 'high')], 'output': 'very_high'},

            # Context-specific patterns
            {'conditions': [('global_uncertainty', 'high')], 'output': 'medium'},
            {'conditions': [('activity', 'high')], 'output': 'medium'},
            {'conditions': [('coupling', 'high')], 'output': 'medium'},
        ]

        # Convert patterns to rule format
        for i, pattern in enumerate(rule_patterns):
            rule = {
                'id': i,
                'conditions': pattern['conditions'],
                'output': pattern['output'],
                'weight': 1.0,
                'performance_history': [],
                'activation_count': 0
            }
            self.rules.append(rule)
            self.rule_weights[i] = 1.0
            self.rule_performance[i] = []

        self.logger.info(f"Created {len(self.rules)} systematic rules")
        return self.rules

    def add_rule(self, conditions: List[Tuple[str, str]], output: str,
initial_weight: float = 0.8) -> int:
        """Add a new rule to the rule base."""
        rule_id = len(self.rules)
        rule = {
            'id': rule_id,
```

```
            'conditions': conditions,
            'output': output,
            'weight': initial_weight,
            'performance_history': [],
            'activation_count': 0
        }

        self.rules.append(rule)
        self.rule_weights[rule_id] = initial_weight
        self.rule_performance[rule_id] = []

        # Record rule generation
        self.rule_generation_history.append({
            'timestamp': time.time(),
            'rule_id': rule_id,
            'conditions': conditions,
            'output': output,
            'initial_weight': initial_weight,
            'generation_type': 'adaptive'
        })

        self.logger.debug(f"Added rule {rule_id}: {conditions} -> {output}")
        return rule_id

    def update_rule_weight(self, rule_id: int, new_weight: float):
        """Update the weight of a specific rule."""
        if 0 <= rule_id < len(self.rules):
            self.rule_weights[rule_id] = np.clip(new_weight, 0.1, 2.0)
            self.rules[rule_id]['weight'] = self.rule_weights[rule_id]

    def record_rule_performance(self, rule_id: int, error: float):
        """Record performance feedback for a rule."""
        if rule_id in self.rule_performance:
            self.rule_performance[rule_id].append(error)
            if len(self.rule_performance[rule_id]) > 20:
                self.rule_performance[rule_id] =
self.rule_performance[rule_id][-20:]

    def prune_underperforming_rules(self) -> int:
        """Remove underperforming rules based on multiple criteria."""
        if len(self.rules) <= self.config.min_rules:
            return 0

        rules_to_remove = []

        for rule_id, performance_history in self.rule_performance.items():
            if len(performance_history) >= 5:
                avg_error = np.mean(performance_history)
                activation_count = self.rule_activations.get(rule_id, 0)
                current_weight = self.rule_weights.get(rule_id, 1.0)
```

```
                # Multi-criteria pruning score
                error_score = 4 if avg_error > 1.5 else (2 if avg_error > 1.0 else 
0)
                activation_score = 3 if activation_count < 3 else (1 if 
activation_count < 10 else 0)
                weight_score = 2 if current_weight < 0.3 else (1 if current_weight < 
0.5 else 0)

                pruning_score = 0.4 * error_score + 0.3 * activation_score + 0.3 * 
weight_score

                if pruning_score >= 3 and len(self.rules) - len(rules_to_remove) > 
self.config.min_rules:
                    rules_to_remove.append(rule_id)

        # Remove rules
        if rules_to_remove:
            self._remove_rules(rules_to_remove)
            self.logger.info(f"Pruned {len(rules_to_remove)} underperforming rules")

        return len(rules_to_remove)

    def _remove_rules(self, indices_to_remove: List[int]):
        """Remove rules and update associated data structures."""
        # Sort in reverse order to maintain indices during removal
        indices_to_remove.sort(reverse=True)

        for rule_id in indices_to_remove:
            if 0 <= rule_id < len(self.rules):
                del self.rules[rule_id]
                if rule_id in self.rule_weights:
                    del self.rule_weights[rule_id]
                if rule_id in self.rule_performance:
                    del self.rule_performance[rule_id]
                if rule_id in self.rule_activations:
                    del self.rule_activations[rule_id]

        # Reindex remaining rules
        self._reindex_rules()

    def _reindex_rules(self):
        """Reindex rules after removal."""
        new_weights = {}
        new_performance = {}
        new_activations = defaultdict(int)

        for new_id, rule in enumerate(self.rules):
            old_id = rule['id']
            rule['id'] = new_id
```

```
            if old_id in self.rule_weights:
                new_weights[new_id] = self.rule_weights[old_id]
            if old_id in self.rule_performance:
                new_performance[new_id] = self.rule_performance[old_id]
            if old_id in self.rule_activations:
                new_activations[new_id] = self.rule_activations[old_id]

        self.rule_weights = new_weights
        self.rule_performance = new_performance
        self.rule_activations = new_activations


class UncertaintyQuantifier:
    """Handles uncertainty quantification using Monte Carlo methods."""

    def __init__(self, config: SOFISConfiguration):
        self.config = config
        self.logger = logging.getLogger(self.__class__.__name__)

    def calculate_aleatoric_uncertainty(self, pose_confidence: float) -> float:
        """Calculate aleatoric (data) uncertainty based on pose confidence."""
        return self.config.uncertainty_alpha * (1.0 - pose_confidence)

    def estimate_epistemic_uncertainty(self, inputs: Dict[str, float],
                                       rule_base: List[Dict],
                                       rule_weights: Dict[int, float],
                                       membership_functions: Dict) ->
Tuple[List[float], float]:
        """Estimate epistemic uncertainty using Monte Carlo dropout."""
        mc_predictions = []

        for _ in range(self.config.mc_samples):
            # Apply dropout to rule weights
            modified_weights = {}
            for rule_id, weight in rule_weights.items():
                if np.random.rand() > self.config.dropout_rate:
                    modified_weights[rule_id] = weight
                else:
                    modified_weights[rule_id] = weight * 0.5

            # Perform inference with modified weights
            try:
                prediction = self._perform_inference(inputs, rule_base,
modified_weights, membership_functions)
                mc_predictions.append(prediction)
            except Exception as e:
                self.logger.debug(f"Monte Carlo sample failed: {e}")
                continue
```

```
        # Calculate epistemic uncertainty
        if len(mc_predictions) > 1:
            epistemic_uncertainty = np.std(mc_predictions)
        else:
            epistemic_uncertainty = 1.0

        return mc_predictions, epistemic_uncertainty

    def _perform_inference(self, inputs: Dict[str, float],
                          rule_base: List[Dict],
                          rule_weights: Dict[int, float],
                          membership_functions: Dict) -> float:
        """Perform fuzzy inference using simplified method, where min access to
computational power"""
        # Simplified fuzzy inference without scikit-fuzzy dependency
        total_activation = 0.0
        weighted_sum = 0.0

        # Output mapping
        output_values = {
            'negligible': 1.5,
            'low': 3.0,
            'medium': 6.0,
            'high': 9.0,
            'very_high': 13.0
        }

        for rule in rule_base:
            rule_id = rule['id']
            weight = rule_weights.get(rule_id, 1.0)

            # Calculate rule activation
            activation = self._calculate_rule_activation(rule, inputs,
membership_functions)

            if activation > 1e-6:
                output_value = output_values.get(rule['output'], 6.0)
                weighted_sum += weight * activation * output_value
                total_activation += weight * activation

        if total_activation > 0:
            return weighted_sum / total_activation
        else:
            return 6.0  # Default medium risk

    def _calculate_rule_activation(self, rule: Dict, inputs: Dict[str, float],
                                  membership_functions: Dict) -> float:
        """Calculate activation strength for a rule."""
        min_activation = 1.0
```

```
        for var_name, term in rule['conditions']:
            if var_name in inputs and var_name in membership_functions:
                value = inputs[var_name]
                mf_params = membership_functions[var_name].get(term, {})
                if mf_params:
                    center = mf_params.get('center', 0)
                    sigma = mf_params.get('sigma', 1)
                    membership = np.exp(-0.5 * ((value - center) / sigma) ** 2)
                    min_activation = min(min_activation, membership)
                else:
                    min_activation = 0.0
                    break
            else:
                min_activation = 0.0
                break

        return min_activation

    def calculate_total_uncertainty(self, aleatoric: float, epistemic: float) ->
float:
        """Calculate total uncertainty by combining components."""
        return np.sqrt(aleatoric**2 + epistemic**2)

    def compute_confidence_interval(self, prediction: float, total_uncertainty:
float,
                                  confidence_level: float = 0.95) -> Tuple[float,
float]:
        """Compute confidence interval for prediction."""
        z_score = stats.norm.ppf((1 + confidence_level) / 2)
        interval_width = z_score * total_uncertainty

        lower_bound = max(1.0, prediction - interval_width)
        upper_bound = min(15.0, prediction + interval_width)

        return lower_bound, upper_bound


class TaskClusterManager:
    """Manages task-specific clustering and specialization."""

    def __init__(self, config: SOFISConfiguration):
        self.config = config
        self.logger = logging.getLogger(self.__class__.__name__)
        self.kmeans_model = None
        self.feature_scaler = None
        self.pca_transformer = None
        self.cluster_labels = None
        self.cluster_centers = None
        self.task_specific_rules = defaultdict(list)
```

```
    def identify_task_clusters(self, training_data: pd.DataFrame,
                             pose_features: Optional[np.ndarray] = None) -> bool:
        """Identify task clusters from training data."""
        if training_data.empty:
            self.logger.error("Cannot cluster: training data is empty")
            return False

        try:
            # Extract features for clustering
            feature_matrix = self._extract_clustering_features(training_data,
pose_features)

            if feature_matrix.shape[1] < 5:
                self.logger.warning("Insufficient features for robust clustering")
                return False

            # Normalize features
            self.feature_scaler = StandardScaler()
            features_scaled = self.feature_scaler.fit_transform(feature_matrix)

            # Apply K-means clustering
            self.kmeans_model = KMeans(
                n_clusters=self.config.n_clusters,
                random_state=self.config.random_seed,
                n_init=10,
                max_iter=300
            )
            self.cluster_labels = self.kmeans_model.fit_predict(features_scaled)
            self.cluster_centers = self.kmeans_model.cluster_centers_

            # Evaluate clustering quality
            silhouette_avg = silhouette_score(features_scaled, self.cluster_labels)

            self.logger.info(f"Task clustering completed: {self.config.n_clusters}
clusters, "
                           f"silhouette score = {silhouette_avg:.3f}")

            return silhouette_avg > self.config.silhouette_threshold

        except Exception as e:
            self.logger.error(f"Task clustering failed: {e}")
            return False

    def _extract_clustering_features(self, training_data: pd.DataFrame,
                                   pose_features: Optional[np.ndarray] = None) ->
np.ndarray:
        """Extract comprehensive features for task clustering."""
        features = []

        # Angle features
```

```
        angle_cols = [col for col in training_data.columns if 'angle' in col]
        if angle_cols:
            angle_features = training_data[angle_cols].fillna(0).values
            features.append(angle_features)

        # Context features
        context_cols = ['load', 'coupling', 'activity']
        context_data = []
        for col in context_cols:
            if col in training_data.columns:
                context_data.append(training_data[col].fillna(0).values)
            else:
                context_data.append(np.zeros(len(training_data)))

        if context_data:
            features.append(np.column_stack(context_data))

        # Uncertainty features
        if 'global_uncertainty' in training_data.columns:
            uncertainty_features =
training_data[['global_uncertainty']].fillna(0.5).values
        else:
            uncertainty_features = np.full((len(training_data), 1), 0.5)

        features.append(uncertainty_features)

        # PCA features from pose estimation if available
        if pose_features is not None and pose_features.shape[0] ==
len(training_data):
            if self.pca_transformer is None:
                self.pca_transformer = PCA(n_components=min(9,
pose_features.shape[1]),
                                         random_state=self.config.random_seed)
                pca_features = self.pca_transformer.fit_transform(pose_features)
            else:
                pca_features = self.pca_transformer.transform(pose_features)
            features.append(pca_features)

        # Combine all features
        if features:
            return np.hstack(features)
        else:
            # Return dummy features if no valid features found
            return np.random.normal(0, 1, (len(training_data), 5))

    def predict_task_cluster(self, features: np.ndarray) -> Optional[int]:
        """Predict task cluster for new features."""
        if self.kmeans_model is None or self.feature_scaler is None:
            return None
```

```
        try:
            features_scaled = self.feature_scaler.transform(features.reshape(1, -1))
            cluster = self.kmeans_model.predict(features_scaled)[0]
            return int(cluster)
        except Exception as e:
            self.logger.warning(f"Failed to predict task cluster: {e}")
            return None


class ExpertFeedbackProcessor:
    """Processes expert feedback for continuous system improvement."""

    def __init__(self, config: SOFISConfiguration):
        self.config = config
        self.logger = logging.getLogger(self.__class__.__name__)
        self.feedback_history = []
        self.learning_curves = defaultdict(list)

    def process_expert_feedback(self, assessment_result: AssessmentResult,
                                expert_score: float, expert_confidence: float = 1.0)
-> Dict[str, Any]:
        """Process expert feedback and generate learning updates."""
        prediction_error = expert_score - assessment_result.risk_score

        # Record feedback
        feedback_record = {
            'timestamp': time.time(),
            'system_prediction': assessment_result.risk_score,
            'expert_score': expert_score,
            'prediction_error': prediction_error,
            'expert_confidence': expert_confidence,
            'system_uncertainty': assessment_result.total_uncertainty,
            'rule_activations': assessment_result.rule_activations.copy()
        }
        self.feedback_history.append(feedback_record)

        # Generate learning updates
        updates = self._generate_learning_updates(feedback_record)

        # Update learning curves
        self.learning_curves['prediction_error'].append(abs(prediction_error))
        self.learning_curves['expert_confidence'].append(expert_confidence)

self.learning_curves['system_uncertainty'].append(assessment_result.total_uncertaint
y)

        self.logger.info(f"Processed expert feedback: error={prediction_error:.3f},
"
                        f"confidence={expert_confidence:.3f}")
```

```
        return updates

    def _generate_learning_updates(self, feedback: Dict[str, Any]) -> Dict[str,
Any]:
        """Generate specific learning updates from feedback."""
        updates = {
            'rule_weight_updates': {},
            'structural_changes': {},
            'learning_priority': self._calculate_learning_priority(feedback)
        }

        prediction_error = feedback['prediction_error']
        expert_confidence = feedback['expert_confidence']
        rule_activations = feedback['rule_activations']

        # Calculate rule weight updates
        if abs(prediction_error) > 0.5:
            for rule_idx, activation in rule_activations.items():
                if activation > 1e-6:
                    weight_update = (self.config.learning_rate * prediction_error *
                                   activation * expert_confidence)
                    updates['rule_weight_updates'][int(rule_idx)] = weight_update

        # Suggest structural changes for large errors
        if abs(prediction_error) > 2.0 and expert_confidence > 0.8:
            updates['structural_changes']['suggest_new_rule'] = True
            updates['structural_changes']['error_magnitude'] = abs(prediction_error)

        return updates

    def _calculate_learning_priority(self, feedback: Dict[str, Any]) -> float:
        """Calculate learning priority based on feedback characteristics."""
        error_magnitude = abs(feedback['prediction_error'])
        expert_confidence = feedback['expert_confidence']
        system_uncertainty = feedback['system_uncertainty']

        priority = (error_magnitude * expert_confidence) / (1.0 +
system_uncertainty)
        return min(priority, 1.0)


class SOFISSystem:
    """
    Main SOFIS system integrating all components for adaptive REBA assessment.
    """

    def __init__(self, config: Optional[SOFISConfiguration] = None, output_dir: str
= "sofis_results"):
        """Initialize the SOFIS system."""
        self.config = config or SOFISConfiguration()
```

```
        self.output_dir = Path(output_dir)
        self.logger = logging.getLogger(self.__class__.__name__)

        # Create output directories
        self._setup_output_directories()

        # Initialize components
        self.pose_processor = PoseProcessor()
        self.membership_manager = MembershipFunctionManager(self.config)
        self.rule_manager = RuleBaseManager(self.config)
        self.uncertainty_quantifier = UncertaintyQuantifier(self.config)
        self.cluster_manager = TaskClusterManager(self.config)
        self.feedback_processor = ExpertFeedbackProcessor(self.config)

        # System state
        self.membership_functions = {}
        self.is_initialized = False
        self.training_data = None
        self.performance_history = []

        # Variable definitions
        self._define_variable_ranges()

        self.logger.info("SOFIS system initialized successfully")

    def _setup_output_directories(self):
        """Create output directory structure."""
        directories = [
            self.output_dir,
            self.output_dir / "data",
            self.output_dir / "figures",
            self.output_dir / "models",
            self.output_dir / "logs"
        ]

        for directory in directories:
            directory.mkdir(parents=True, exist_ok=True)

    def _define_variable_ranges(self):
        """Define ranges for all 19 input variables."""
        self.variable_ranges = {
            # Joint angles (15 variables)
            'trunk_angle': (0, 120),
            'neck_angle': (0, 80),
            'shoulder_flex_angle': (0, 180),
            'shoulder_abd_angle': (0, 180),
            'upper_arm_angle': (0, 180),
            'elbow_angle': (0, 180),
            'lower_arm_angle': (0, 180),
            'wrist_flex_angle': (0, 90),
```

```python
            'wrist_dev_angle': (0, 90),
            'hip_angle': (0, 120),
            'knee_angle': (0, 160),
            'ankle_angle': (0, 90),
            'leg_angle': (0, 160),
            'spine_angle': (0, 90),
            'head_angle': (0, 60),

            # Context variables (3 variables)
            'load': (0, 10),
            'coupling': (0, 3),
            'activity': (0, 3),

            # Global uncertainty (1 variable)
            'global_uncertainty': (0, 1)
        }

    def initialize_system(self, training_data: pd.DataFrame) -> bool:
        """Initialize the complete SOFIS system with training data."""
        try:
            self.logger.info("Initializing SOFIS system...")
            self.training_data = training_data.copy()

            # Validate training data
            self._validate_training_data(training_data)

            # Learn membership functions
            self._learn_membership_functions(training_data)

            # Initialize rule base
            self.rule_manager.initialize_systematic_rules()

            # Identify task clusters
            self.cluster_manager.identify_task_clusters(training_data)

            # Generate visualizations
            self._generate_initialization_visualizations()

            self.is_initialized = True
            self.logger.info("SOFIS system initialization completed successfully")
            return True

        except Exception as e:
            self.logger.error(f"SOFIS initialization failed: {e}")
            return False

    def _validate_training_data(self, training_data: pd.DataFrame):
        """Validate training data format and content."""
        required_columns = ['trunk_angle', 'neck_angle', 'upper_arm_angle', 'load']
        missing_columns = [col for col in required_columns if col not in
```

```
training_data.columns]

        if missing_columns:
            raise DataValidationError(f"Missing required columns:
{missing_columns}")

        if len(training_data) < 10:
            raise DataValidationError("Insufficient training data (minimum 10
samples required)")

    def _learn_membership_functions(self, training_data: pd.DataFrame):
        """Learn membership functions for all variables from training data."""
        self.logger.info("Learning membership functions from training data...")

        for var_name, (min_val, max_val) in self.variable_ranges.items():
            if var_name in training_data.columns:
                data = training_data[var_name].dropna().values
                if len(data) > 0:
                    # Ensure data is within expected range
                    data = np.clip(data, min_val, max_val)
                    self.membership_functions[var_name] =
self.membership_manager.learn_membership_functions(data, var_name)
                else:
                    self.membership_functions[var_name] =
self.membership_manager._create_default_functions(min_val, max_val)
            else:
                self.membership_functions[var_name] =
self.membership_manager._create_default_functions(min_val, max_val)

        self.logger.info(f"Learned membership functions for
{len(self.membership_functions)} variables")

    def assess_risk(self, pose_data: Optional[Dict[str, np.ndarray]] = None,
                   joint_angles: Optional[Dict[str, float]] = None,
                   context: Optional[Dict[str, Any]] = None,
                   pose_confidences: Optional[Dict[str, float]] = None) ->
AssessmentResult:
        """Perform comprehensive REBA risk assessment."""
        start_time = time.time()

        if not self.is_initialized:
            raise FuzzySystemError("SOFIS system not initialized. Call
initialize_system() first.")

        try:
            # Step 1: Process pose data to get joint angles
            if joint_angles is None:
                if pose_data is None:
                    raise ValueError("Either pose_data or joint_angles must be
provided")
```

```
                joint_angles = self.pose_processor.calculate_joint_angles(pose_data)

            # Step 2: Prepare inputs
            validated_inputs = self._prepare_inputs(joint_angles, context,
pose_confidences)

            # Step 3: Calculate global uncertainty
            global_uncertainty =
self._calculate_global_uncertainty(pose_confidences)
            validated_inputs['global_uncertainty'] = global_uncertainty

            # Step 4: Predict task cluster
            feature_vector = self._extract_feature_vector(validated_inputs)
            task_cluster = self.cluster_manager.predict_task_cluster(feature_vector)

            # Step 5: Calculate aleatoric uncertainty
            pose_confidence_avg = np.mean(list(pose_confidences.values())) if
pose_confidences else 0.8
            aleatoric_uncertainty =
self.uncertainty_quantifier.calculate_aleatoric_uncertainty(pose_confidence_avg)

            # Step 6: Perform primary inference
            primary_result = self._perform_inference(validated_inputs)

            # Step 7: Estimate epistemic uncertainty
            mc_predictions, epistemic_uncertainty =
self.uncertainty_quantifier.estimate_epistemic_uncertainty(
                validated_inputs, self.rule_manager.rules,
self.rule_manager.rule_weights, self.membership_functions
            )

            # Step 8: Calculate total uncertainty and confidence
            total_uncertainty =
self.uncertainty_quantifier.calculate_total_uncertainty(
                aleatoric_uncertainty, epistemic_uncertainty
            )

            confidence_interval =
self.uncertainty_quantifier.compute_confidence_interval(
                primary_result, total_uncertainty
            )

            confidence_score = max(0.0, min(1.0, 1.0 - (total_uncertainty / 4.0)))

            # Step 9: Generate results
            risk_category = self._categorize_risk(primary_result)

            processing_time = time.time() - start_time

            result = AssessmentResult(
```

```
                risk_score=primary_result,
                risk_category=risk_category,
                confidence_score=confidence_score,
                uncertainty_interval=confidence_interval,
                aleatoric_uncertainty=aleatoric_uncertainty,
                epistemic_uncertainty=epistemic_uncertainty,
                total_uncertainty=total_uncertainty,
                rule_activations={},  # Simplified for this implementation
                input_features=validated_inputs,
                task_cluster=task_cluster,
                explanation=self._generate_explanation(primary_result,
validated_inputs),
                processing_time=processing_time
            )

            # Track performance
            self._track_assessment_performance(result)

            return result

        except Exception as e:
            self.logger.error(f"Risk assessment failed: {e}")
            return self._generate_fallback_result(joint_angles or {}, context,
str(e))

    def _prepare_inputs(self, joint_angles: Dict[str, float],
                       context: Optional[Dict[str, Any]],
                       pose_confidences: Optional[Dict[str, float]]) -> Dict[str,
float]:
        """Prepare and validate inputs for assessment."""
        # Set default context
        if context is None:
            context = {}

        context.setdefault('load', 0.0)
        context.setdefault('coupling', 0)
        context.setdefault('activity', 0)

        # Combine all inputs
        all_inputs = {**joint_angles, **context}

        # Validate and fill missing inputs
        validated_inputs = {}
        for var_name, (min_val, max_val) in self.variable_ranges.items():
            if var_name == 'global_uncertainty':
                continue  # Handled separately

            if var_name in all_inputs and all_inputs[var_name] is not None:
                value = float(all_inputs[var_name])
                validated_inputs[var_name] = np.clip(value, min_val, max_val)
```

```
            else:
                # Use intelligent defaults
                default_value = self._get_default_value(var_name, min_val, max_val)
                validated_inputs[var_name] = default_value
                self.logger.debug(f"Using default for {var_name}: {default_value}")

        return validated_inputs

    def _get_default_value(self, var_name: str, min_val: float, max_val: float) ->
float:
        """Get intelligent default values for missing inputs."""
        defaults = {
            'trunk_angle': 10.0,
            'neck_angle': 5.0,
            'upper_arm_angle': 20.0,
            'elbow_angle': 90.0,
            'wrist_flex_angle': 5.0,
            'load': 0.0,
            'coupling': 1,
            'activity': 0
        }

        return defaults.get(var_name, (min_val + max_val) / 2.0)

    def _calculate_global_uncertainty(self, pose_confidences: Optional[Dict[str,
float]]) -> float:
        """Calculate global uncertainty from pose confidences."""
        if pose_confidences is None or not pose_confidences:
            return 0.5

        confidences = [c for c in pose_confidences.values() if c is not None]
        if not confidences:
            return 0.5

        return np.clip(1.0 - np.mean(confidences), 0.0, 1.0)

    def _extract_feature_vector(self, inputs: Dict[str, float]) -> np.ndarray:
        """Extract feature vector for task clustering."""
        features = []

        # Key angle features
        angle_vars = ['trunk_angle', 'neck_angle', 'upper_arm_angle',
'lower_arm_angle',
                     'wrist_flex_angle', 'leg_angle']
        for var in angle_vars:
            features.append(inputs.get(var, 0.0))

        # Context features
        context_vars = ['load', 'coupling', 'activity']
        for var in context_vars:
```

```
            features.append(inputs.get(var, 0.0))

        # Uncertainty feature
        features.append(inputs.get('global_uncertainty', 0.5))

        return np.array(features)

    def _perform_inference(self, inputs: Dict[str, float]) -> float:
        """Perform fuzzy inference using the rule base."""
        return self.uncertainty_quantifier._perform_inference(
            inputs, self.rule_manager.rules, self.rule_manager.rule_weights,
self.membership_functions
        )

    def _categorize_risk(self, score: float) -> str:
        """Convert REBA score to risk category."""
        if score < 2:
            return "Negligible"
        elif score < 4:
            return "Low"
        elif score < 8:
            return "Medium"
        elif score < 11:
            return "High"
        else:
            return "Very High"

    def _generate_explanation(self, risk_score: float, inputs: Dict[str, float]) ->
str:
        """Generate human-readable explanation of the assessment."""
        risk_factors = []

        if inputs.get('trunk_angle', 0) > 60:
            risk_factors.append("high trunk flexion")
        if inputs.get('upper_arm_angle', 0) > 90:
            risk_factors.append("elevated arm position")
        if inputs.get('load', 0) > 8:
            risk_factors.append("heavy load")
        if inputs.get('coupling', 0) > 2:
            risk_factors.append("poor coupling")

        explanation = f"REBA score {risk_score:.1f}"
        if risk_factors:
            explanation += f" based on key factors: {', '.join(risk_factors)}"

        return explanation

    def _track_assessment_performance(self, result: AssessmentResult):
        """Track assessment performance for monitoring."""
        performance_record = {
```

```
            'timestamp': result.timestamp,
            'risk_score': result.risk_score,
            'risk_category': result.risk_category,
            'confidence_score': result.confidence_score,
            'total_uncertainty': result.total_uncertainty,
            'task_cluster': result.task_cluster,
            'processing_time': result.processing_time
        }

        self.performance_history.append(performance_record)

        # Keep only recent history
        if len(self.performance_history) > 1000:
            self.performance_history = self.performance_history[-1000:]

    def _generate_fallback_result(self, joint_angles: Dict, context: Optional[Dict],

                                error: str) -> AssessmentResult:
        """Generate fallback result when assessment fails."""
        self.logger.warning(f"Generating fallback assessment: {error}")

        # Simple heuristic
        risk_score = 5.0
        try:
            if joint_angles.get('trunk_angle', 0) > 60:
                risk_score += 2
            if joint_angles.get('upper_arm_angle', 0) > 90:
                risk_score += 1
            if context and context.get('load', 0) > 8:
                risk_score += 2
        except:
            pass

        risk_score = np.clip(risk_score, 1.0, 15.0)

        return AssessmentResult(
            risk_score=float(risk_score),
            risk_category=self._categorize_risk(risk_score),
            confidence_score=0.3,
            uncertainty_interval=(max(1.0, risk_score-3), min(15.0, risk_score+3)),
            aleatoric_uncertainty=2.0,
            epistemic_uncertainty=2.0,
            total_uncertainty=2.8,
            rule_activations={},
            input_features=joint_angles,
            explanation=f"Fallback assessment due to error: {error}",
            is_fallback=True
        )

    def learn_from_expert(self, assessment_result: AssessmentResult,
```

```
                         expert_score: float, expert_confidence: float = 1.0) ->
bool:
        """Learn from expert feedback to improve system performance."""
        try:
            # Process expert feedback
            updates = self.feedback_processor.process_expert_feedback(
                assessment_result, expert_score, expert_confidence
            )

            # Apply rule weight updates
            for rule_idx, weight_update in updates['rule_weight_updates'].items():
                current_weight = self.rule_manager.rule_weights.get(rule_idx, 1.0)
                new_weight = current_weight + weight_update
                self.rule_manager.update_rule_weight(rule_idx, new_weight)

                # Record performance feedback
                error = abs(expert_score - assessment_result.risk_score)
                self.rule_manager.record_rule_performance(rule_idx, error)

            # Apply structural changes if suggested
            if updates['structural_changes'].get('suggest_new_rule', False):
                self._generate_new_rule_from_feedback(assessment_result,
expert_score)

            # Periodic rule pruning
            if len(self.feedback_processor.feedback_history) % 15 == 0:
                pruned_count = self.rule_manager.prune_underperforming_rules()
                if pruned_count > 0:
                    self.logger.info(f"Pruned {pruned_count} underperforming rules")

            self.logger.info(f"Learning completed:
{len(updates['rule_weight_updates'])} rule updates applied")
            return True

        except Exception as e:
            self.logger.error(f"Expert learning failed: {e}")
            return False

    def _generate_new_rule_from_feedback(self, assessment: AssessmentResult,
expert_score: float):
        """Generate a new rule based on expert feedback."""
        try:
            inputs = assessment.input_features

            # Select discriminative variables
            conditions = []
            key_variables = ['trunk_angle', 'upper_arm_angle', 'neck_angle', 'load']

            for var_name in key_variables:
                if var_name in inputs and var_name in self.membership_functions:
```

```
                    value = inputs[var_name]
                    best_term = self._find_best_membership_term(var_name, value)
                    if best_term:
                        conditions.append((var_name, best_term))

            # Map score to output term
            output_term = self._map_score_to_output_term(expert_score)

            if len(conditions) >= 2 and output_term:
                rule_id = self.rule_manager.add_rule(conditions, output_term,
initial_weight=0.8)
                self.logger.info(f"Generated new rule {rule_id} from expert
feedback")

        except Exception as e:
            self.logger.warning(f"Failed to generate new rule: {e}")

    def _find_best_membership_term(self, var_name: str, value: float) ->
Optional[str]:
        """Find the term with highest membership for a value."""
        if var_name not in self.membership_functions:
            return None

        best_term = None
        max_membership = -1.0

        for term_name, mf_params in self.membership_functions[var_name].items():
            membership = self.membership_manager.get_membership_value(value,
mf_params)
            if membership > max_membership:
                max_membership = membership
                best_term = term_name

        return best_term if max_membership > 0.1 else None

    def _map_score_to_output_term(self, score: float) -> Optional[str]:
        """Map REBA score to appropriate output term."""
        if score < 2:
            return 'negligible'
        elif score < 4:
            return 'low'
        elif score < 8:
            return 'medium'
        elif score < 11:
            return 'high'
        else:
            return 'very_high'

    def evaluate_performance(self, test_data: pd.DataFrame) -> Dict[str, Any]:
        """Evaluate system performance on test data."""
```

```
        if not self.is_initialized:
            raise FuzzySystemError("System not initialized")

        predictions = []
        true_scores = []
        processing_times = []

        for _, row in test_data.iterrows():
            try:
                # Extract joint angles
                joint_angles = {col: row[col] for col in test_data.columns
                               if 'angle' in col and col in self.variable_ranges}

                # Extract context
                context = {}
                for col in ['load', 'coupling', 'activity']:
                    if col in test_data.columns:
                        context[col] = row[col]

                # Perform assessment
                result = self.assess_risk(joint_angles=joint_angles,
context=context)

                predictions.append(result.risk_score)
                processing_times.append(result.processing_time)

                if 'expert_reba_score' in test_data.columns:
                    true_scores.append(row['expert_reba_score'])

            except Exception as e:
                self.logger.warning(f"Failed to assess sample: {e}")
                continue

        # Calculate metrics
        metrics = {
            'n_samples': len(predictions),
            'mean_prediction': np.mean(predictions),
            'std_prediction': np.std(predictions),
            'mean_processing_time': np.mean(processing_times),
            'std_processing_time': np.std(processing_times)
        }

        if true_scores:
            mae = mean_absolute_error(true_scores, predictions)
            rmse = np.sqrt(mean_squared_error(true_scores, predictions))

            # Calculate accuracy for categorical prediction
            true_categories = [self._categorize_risk(score) for score in
true_scores]
            pred_categories = [self._categorize_risk(score) for score in
```

```
predictions]

            correct_predictions = sum(1 for t, p in zip(true_categories,
pred_categories) if t == p)
            accuracy = correct_predictions / len(true_categories)

            metrics.update({
                'mae': mae,
                'rmse': rmse,
                'accuracy': accuracy,
                'correlation': np.corrcoef(true_scores, predictions)[0, 1] if
len(predictions) > 1 else 0
            })

        return metrics

    def _generate_initialization_visualizations(self):
        """Generate visualizations after system initialization."""
        try:
            if self.training_data is not None:
                self._plot_training_data_distribution()
                self._plot_membership_functions()

            self.logger.info("Initialization visualizations generated")

        except Exception as e:
            self.logger.warning(f"Failed to generate visualizations: {e}")

    def _plot_training_data_distribution(self):
        """Plot distribution of training data."""
        try:
            fig, axes = plt.subplots(2, 3, figsize=(15, 10))
            axes = axes.flatten()

            key_variables = ['trunk_angle', 'neck_angle', 'upper_arm_angle', 'load',
'coupling', 'activity']

            for i, var in enumerate(key_variables):
                if var in self.training_data.columns and i < len(axes):
                    axes[i].hist(self.training_data[var].dropna(), bins=20,
alpha=0.7, edgecolor='black')
                    axes[i].set_title(f'Distribution of {var}')
                    axes[i].set_xlabel(var)
                    axes[i].set_ylabel('Frequency')
                    axes[i].grid(True, alpha=0.3)

            # Remove unused subplots
            for i in range(len(key_variables), len(axes)):
                fig.delaxes(axes[i])
```

```
            plt.tight_layout()
            plt.savefig(self.output_dir / "figures" /
"training_data_distribution.png",
                       dpi=300, bbox_inches='tight')
            plt.close()

        except Exception as e:
            self.logger.warning(f"Error plotting training data distribution: {e}")

    def _plot_membership_functions(self):
        """Plot learned membership functions."""
        try:
            key_variables = ['trunk_angle', 'upper_arm_angle', 'load',
'global_uncertainty']

            fig, axes = plt.subplots(2, 2, figsize=(12, 10))
            axes = axes.flatten()

            for i, var_name in enumerate(key_variables):
                if var_name in self.membership_functions and i < len(axes):
                    ax = axes[i]

                    min_val, max_val = self.variable_ranges[var_name]
                    x = np.linspace(min_val, max_val, 100)

                    for term_name, mf_params in
self.membership_functions[var_name].items():
                        center = mf_params['center']
                        sigma = mf_params['sigma']
                        y = np.exp(-0.5 * ((x - center) / sigma) ** 2)
                        ax.plot(x, y, label=term_name, linewidth=2)

                    ax.set_title(f'Membership Functions: {var_name}')
                    ax.set_xlabel('Input Value')
                    ax.set_ylabel('Membership Degree')
                    ax.legend()
                    ax.grid(True, alpha=0.3)

            plt.tight_layout()
            plt.savefig(self.output_dir / "figures" / "membership_functions.png",
                       dpi=300, bbox_inches='tight')
            plt.close()

        except Exception as e:
            self.logger.warning(f"Error plotting membership functions: {e}")

    def save_model(self, filepath: Optional[str] = None) -> bool:
        """Save the complete SOFIS model to disk."""
        if filepath is None:
            timestamp = time.strftime("%Y%m%d_%H%M%S")
```

```
            filepath = self.output_dir / "models" / f"sofis_model_{timestamp}.pkl"

        try:
            model_state = {
                'config': asdict(self.config),
                'membership_functions': self.membership_functions,
                'rules': self.rule_manager.rules,
                'rule_weights': self.rule_manager.rule_weights,
                'rule_performance': self.rule_manager.rule_performance,
                'cluster_model': {
                    'kmeans_centers': self.cluster_manager.cluster_centers.tolist()
                                    if self.cluster_manager.cluster_centers is not
None else None,
                    'cluster_labels': self.cluster_manager.cluster_labels.tolist()
                                    if self.cluster_manager.cluster_labels is not
None else None,
                    'feature_scaler': self.cluster_manager.feature_scaler,
                    'pca_transformer': self.cluster_manager.pca_transformer
                },
                'feedback_history': self.feedback_processor.feedback_history,
                'performance_history': self.performance_history,
                'variable_ranges': self.variable_ranges,
                'timestamp': time.time()
            }

            with open(filepath, 'wb') as f:
                pickle.dump(model_state, f, protocol=pickle.HIGHEST_PROTOCOL)

            self.logger.info(f"Model saved successfully to {filepath}")
            return True

        except Exception as e:
            self.logger.error(f"Failed to save model: {e}")
            return False

    def load_model(self, filepath: str) -> bool:
        """Load a previously saved SOFIS model."""
        try:
            with open(filepath, 'rb') as f:
                model_state = pickle.load(f)

            # Restore configuration
            self.config = SOFISConfiguration(**model_state['config'])

            # Restore system state
            self.membership_functions = model_state.get('membership_functions', {})
            self.rule_manager.rules = model_state.get('rules', [])
            self.rule_manager.rule_weights = model_state.get('rule_weights', {})
            self.rule_manager.rule_performance = model_state.get('rule_performance',
{})
```

```
            # Restore other components
            cluster_data = model_state.get('cluster_model', {})
            if cluster_data.get('cluster_labels') is not None:
                self.cluster_manager.cluster_labels =
np.array(cluster_data['cluster_labels'])
            if cluster_data.get('kmeans_centers') is not None:
                self.cluster_manager.cluster_centers =
np.array(cluster_data['kmeans_centers'])

            self.cluster_manager.feature_scaler = cluster_data.get('feature_scaler')
            self.cluster_manager.pca_transformer =
cluster_data.get('pca_transformer')

            self.feedback_processor.feedback_history =
model_state.get('feedback_history', [])
            self.performance_history = model_state.get('performance_history', [])
            self.variable_ranges = model_state.get('variable_ranges',
self.variable_ranges)

            self.is_initialized = len(self.membership_functions) > 0 and
len(self.rule_manager.rules) > 0

            self.logger.info(f"Model loaded successfully from {filepath}")
            return True

        except Exception as e:
            self.logger.error(f"Failed to load model: {e}")
            return False

    def get_system_status(self) -> Dict[str, Any]:
        """Get comprehensive system status and statistics."""
        status = {
            'is_initialized': self.is_initialized,
            'config': asdict(self.config),
            'total_rules': len(self.rule_manager.rules),
            'total_assessments': len(self.performance_history),
            'membership_functions_count': len(self.membership_functions),
            'has_clustering': self.cluster_manager.cluster_labels is not None,
            'feedback_sessions': len(self.feedback_processor.feedback_history)
        }

        if self.performance_history:
            recent_performance = self.performance_history[-10:]
            status['recent_performance'] = {
                'avg_processing_time': np.mean([p['processing_time'] for p in
recent_performance]),
                'avg_confidence': np.mean([p['confidence_score'] for p in
recent_performance]),
                'risk_distribution': {
```

```
                    category: sum(1 for p in recent_performance if
p['risk_category'] == category)
                    for category in ['Negligible', 'Low', 'Medium', 'High', 'Very
High']
                }
            }

        return status


# For exmaple usage and testing; you can try with random logical body angles
if __name__ == "__main__":
    # Configure logging for demonstration
    logging.getLogger().setLevel(logging.INFO)

    print("=" * 80)
    print("SOFIS Production System Demonstration")
    print("=" * 80)

    # Initialize system
    config = SOFISConfiguration(
        max_rules=150,
        dropout_rate=0.3,
        mc_samples=10,
        learning_rate=0.05
    )

    sofis = SOFISSystem(config, output_dir="sofis_production_results")

    # Create sample training data
    np.random.seed(42)
    n_samples = 100

    training_data = pd.DataFrame({
        'trunk_angle': np.random.uniform(0, 90, n_samples),
        'neck_angle': np.random.uniform(0, 60, n_samples),
        'upper_arm_angle': np.random.uniform(0, 150, n_samples),
        'lower_arm_angle': np.random.uniform(60, 120, n_samples),
        'wrist_flex_angle': np.random.uniform(0, 45, n_samples),
        'leg_angle': np.random.uniform(0, 90, n_samples),
        'load': np.random.uniform(0, 10, n_samples),
        'coupling': np.random.randint(0, 4, n_samples),
        'activity': np.random.randint(0, 4, n_samples),
        'expert_reba_score': np.random.uniform(1, 12, n_samples),
        'task_type': np.random.choice(['concrete', 'steel', 'roofing'], n_samples)
    })

    # Initialize system
    if sofis.initialize_system(training_data):
        print("SOFIS system initialized successfully")
```

```
        # Example assessment with joint angles
        test_angles = {
            'trunk_angle': 45.0,
            'neck_angle': 25.0,
            'upper_arm_angle': 85.0,
            'lower_arm_angle': 95.0,
            'wrist_flex_angle': 20.0,
            'leg_angle': 30.0
        }

        test_context = {
            'load': 6.0,
            'coupling': 2,
            'activity': 1
        }

        test_confidences = {
            'trunk': 0.85,
            'arms': 0.75,
            'legs': 0.80
        }

        print("\n--- Risk Assessment Demonstration ---")
        result = sofis.assess_risk(
            joint_angles=test_angles,
            context=test_context,
            pose_confidences=test_confidences
        )

        print(f"Risk Score: {result.risk_score:.2f}")
        print(f"Risk Category: {result.risk_category}")
        print(f"Confidence: {result.confidence_score:.3f}")
        print(f"Uncertainty Interval: [{result.uncertainty_interval[0]:.2f},
{result.uncertainty_interval[1]:.2f}]")
        print(f"Processing Time: {result.processing_time:.3f}s")
        print(f"Explanation: {result.explanation}")

        # Demonstrate expert learning
        print("\n--- Expert Learning Demonstration ---")
        expert_score = 8.5
        learning_success = sofis.learn_from_expert(result, expert_score,
expert_confidence=0.9)
        print(f"Expert learning: {'Success' if learning_success else 'Failed'}")

        # Evaluate performance
        print("\n--- Performance Evaluation ---")
        performance = sofis.evaluate_performance(training_data)

        if 'mae' in performance:
```

```
        print(f"MAE: {performance['mae']:.3f}")
        print(f"RMSE: {performance['rmse']:.3f}")
        print(f"Accuracy: {performance['accuracy']:.3f}")

    print(f"Mean Processing Time: {performance['mean_processing_time']:.3f}s")

    # Save model
    if sofis.save_model():
        print("Model saved successfully")

    # Get system status
    print("\n--- System Status ---")
    status = sofis.get_system_status()
    print(f"Initialized: {status['is_initialized']}")
    print(f"Total Rules: {status['total_rules']}")
    print(f"Total Assessments: {status['total_assessments']}")
    print(f"Has Clustering: {status['has_clustering']}")

    print("\n" + "=" * 80)
    print("SOFIS Production System Demo Complete")
    print("=" * 80)

else:
    print("SOFIS system initialization failed")
```