%% The main module program references of this paper are as follows.

```
%% Main Script: Hybrid Transformer-LSTM with BKA Optimization and Fuzzy Early Warning

% This script demonstrates the complete pipeline:

% 1. Load and preprocess experimental task frequency data.

% 2. Define the hybrid Transformer-LSTM model architecture.

% 3. Optimize hyperparameters using the Black-winged Kite Algorithm (BKA).

% 4. Train the final model and evaluate on test data.

% 5. Apply fuzzy comprehensive evaluation for graded early warning.


clear; clc; rng(42); % For reproducibility


%% ==================== 1. Data Loading and Preprocessing ====================

% Load raw multivariate time series (example: synthetic data for illustration)

% In practice, replace with your actual data.

data = load('experimental_task_data.mat'); % Assumes a struct with fields: data.T (time steps), data.X (features)

rawData = data.X; % size: [T, M] where T = number of time steps, M = number of features

targetFeatureIndex = 1; % Assume task frequency is the first feature


% Preprocess data using the function below

[X_train, y_train, X_val, y_val, X_test, y_test, normParams] = preprocessData(rawData, targetFeatureIndex, 0.7, 0.15, 24, 1);

% Inputs: rawData, targetIdx, trainRatio, valRatio, lookBack, horizon

% Outputs: training/validation/test sequences and targets, normalization parameters


%% ==================== 2. BKA Hyperparameter Optimization ====================

% Define search space for hyperparameters:

% [learning rate, LSTM hidden units, number of attention heads, L2 regularization]

lb = [1e-4, 32, 2, 1e-5];    % lower bounds

ub = [1e-2, 128, 8, 1e-2]; % upper bounds

dim = length(lb);


% BKA parameters
```

```
Npop = 20;                  % Swarm size
MaxIter = 50;               % Maximum iterations
pa_init = 0.9;              % Initial attacking probability
pa_final = 0.4;             % Final attacking probability
sigma_cauchy = 0.1; % Cauchy mutation scale
lambda_leader = 0.6;% Leadership coefficient
eta_elite = 0.05;       % Elite perturbation factor
tau_stagnation = 5; % Stagnation tolerance


% Run BKA optimization
[bestParams, bestFitness, convCurve] = BKA_optimizer(@(params)
objectiveFunction(params, X_train, y_train, X_val, y_val), ...
        lb, ub, Npop, MaxIter, pa_init, pa_final, sigma_cauchy, lambda_leader, eta_elite,
tau_stagnation);


fprintf('Optimal hyperparameters:\n');
fprintf('Learning rate: %e\n', bestParams(1));
fprintf('LSTM hidden units: %d\n', round(bestParams(2)));
fprintf('Attention heads: %d\n', round(bestParams(3)));
fprintf('L2 regularization: %e\n', bestParams(4));


%% ===================== 3. Train Final Model with Optimal Params
==============
finalModel = trainHybridModel(X_train, y_train, X_val, y_val, bestParams);
% The model is saved as a struct containing the trained network and options.


%% ===================== 4. Test Set Evaluation
================================
y_pred = predictHybridModel(finalModel, X_test);
testActual = denormalize(y_test, normParams.targetMin, normParams.targetMax);
testPred = denormalize(y_pred, normParams.targetMin, normParams.targetMax);


% Compute error metrics
rmse = sqrt(mean((testActual - testPred).^2));
mae = mean(abs(testActual - testPred));
mape = mean(abs((testActual - testPred) ./ testActual)) * 100;
```

```
fprintf('\nTest Set Performance:\nRMSE = %.4f, MAE = %.4f, MAPE = %.2f%%\n', rmse, mae, mape);


%% ====================== 5. Fuzzy Comprehensive Early Warning ==================

% Compute prediction residuals on test set

residuals = testActual - testPred;


% Generate graded warnings

warningLevels = fuzzyEarlyWarning(residuals, testActual);

% warningLevels is a vector of integers 1..4 indicating Safe, Mild, Moderate, Severe


% Optionally plot the warning levels together with actual frequency

plotEarlyWarning(testActual, testPred, warningLevels);


%% ====================== Helper Functions ========================================


% --------------------------------------------------------------------------

function [X_train, y_train, X_val, y_val, X_test, y_test, normParams] = preprocessData(rawData, targetIdx, trainRatio, valRatio, lookBack, horizon)

    % Preprocess multivariate time series:

    % - Missing value imputation (linear interpolation + forward fill + moving average)

    % - Outlier detection and correction (IQR rule)

    % - Min-max normalization

    % - Sliding window sequence construction

    %

    % Inputs:

    %     rawData      : T x M matrix of raw observations

    %     targetIdx : index of the target variable (task frequency)

    %     trainRatio: fraction of data for training

    %     valRatio    : fraction for validation

    %     lookBack    : number of past steps used as input (L)

    %     horizon      : forecast horizon (h)

    % Outputs:

    %     X_train, y_train, X_val, y_val, X_test, y_test: sequences and targets
```

```
%      normParams: struct with min/max of features for denormalization

[T, M] = size(rawData);

% --- Step 1: Missing value imputation ---
dataImp = rawData;
for m = 1:M
    col = rawData(:, m);
    % Find missing indices (assume NaN indicates missing)
    missIdx = isnan(col);
    if any(missIdx)
        % Linear interpolation for short gaps (max 3 consecutive missing)
        col = fillmissing(col, 'linear', 'MaxGap', 3);
        % Forward fill for remaining long gaps
        col = fillmissing(col, 'previous');
        % Apply moving average smoothing (window = 5)
        col = smoothdata(col, 'movmean', 5);
    end
    dataImp(:, m) = col;
end

% --- Step 2: Outlier detection and correction (IQR) ---
dataClean = dataImp;
for m = 1:M
    col = dataImp(:, m);
    Q1 = prctile(col, 25);
    Q3 = prctile(col, 75);
    IQR = Q3 - Q1;
    lowerBound = Q1 - 1.5 * IQR;
    upperBound = Q3 + 1.5 * IQR;
    outlierIdx = (col < lowerBound) | (col > upperBound);
    if any(outlierIdx)
        % Replace outliers with median of a local window (window size = 5)
        for t = find(outlierIdx)'
            winStart = max(1, t-2);
```

```
                winEnd = min(T, t+2);
                window = col(winStart:winEnd);
                window = window(~outlierIdx(winStart:winEnd)); % exclude other
outliers
                if ~isempty(window)
                    col(t) = median(window);
                else
                    % if window full of outliers, use global median
                    col(t) = median(col(~outlierIdx));
                end
            end
        end
        dataClean(:, m) = col;
    end

    % --- Step 3: Min-max normalization (fit on training portion only) ---
    trainEnd = floor(T * trainRatio);
    valEnd = trainEnd + floor(T * valRatio);
    trainData = dataClean(1:trainEnd, :);
    valData = dataClean(trainEnd+1:valEnd, :);
    testData = dataClean(valEnd+1:end, :);

    % Compute min and max from training set
    featureMin = min(trainData, [], 1);
    featureMax = max(trainData, [], 1);
    % Avoid division by zero if min==max
    featureRange = featureMax - featureMin;
    featureRange(featureRange == 0) = 1;

    % Normalize all sets
    trainNorm = (trainData - featureMin) ./ featureRange;
    valNorm = (valData - featureMin) ./ featureRange;
    testNorm = (testData - featureMin) ./ featureRange;

    % Save normalization parameters for later denormalization
```

```
    normParams.featureMin = featureMin;
    normParams.featureMax = featureMax;
    normParams.targetMin = featureMin(targetIdx);
    normParams.targetMax = featureMax(targetIdx);

    % --- Step 4: Sliding window sequence construction ---
    % Prepare sequences for training
    [X_train, y_train] = createSequences(trainNorm, targetIdx, lookBack, horizon);
    [X_val, y_val] = createSequences(valNorm, targetIdx, lookBack, horizon);
    [X_test, y_test] = createSequences(testNorm, targetIdx, lookBack, horizon);
end

% --------------------------------------------------------------------
function [X, y] = createSequences(dataNorm, targetIdx, lookBack, horizon)
    % dataNorm: N x M normalized matrix
    % Returns:
    %     X: (N - lookBack - horizon + 1) x lookBack x M (or cell array for LSTM input)
    %     y: (N - lookBack - horizon + 1) x 1 target values
    N = size(dataNorm, 1);
    numSamples = N - lookBack - horizon + 1;
    X = cell(numSamples, 1);      % Use cell array for variable-length sequences (though all same length here)
    y = zeros(numSamples, 1);
    for i = 1:numSamples
        X{i} = dataNorm(i:i+lookBack-1, :);      % lookBack x M matrix
        y(i) = dataNorm(i+lookBack+horizon-1, targetIdx);
    end
end

% --------------------------------------------------------------------
function loss = objectiveFunction(params, X_train, y_train, X_val, y_val)
    % Objective function for BKA: train model with given hyperparameters and return validation MSE.
    % params = [learningRate, lstmUnits, numHeads, l2reg]
    learningRate = params(1);
    lstmUnits = round(params(2));
```

```
    numHeads = round(params(3));
    l2reg = params(4);

    % Define network architecture
    layers = buildHybridTransformerLSTM(lstmUnits, numHeads, size(X_train{1},2),
l2reg);

    % Training options
    options = trainingOptions('adam', ...
        'InitialLearnRate', learningRate, ...
        'MaxEpochs', 100, ...
        'MiniBatchSize', 32, ...
        'ValidationData', {X_val, y_val}, ...
        'ValidationFrequency', 30, ...
        'Verbose', false, ...
        'Plots', 'none');

    % Train network
    try
        net = trainNetwork(X_train, y_train, layers, options);
        % Predict on validation set
        y_val_pred = predict(net, X_val);
        loss = mean((y_val - y_val_pred).^2);
    catch ME
        % If training fails, return a large loss
        loss = 1e6;
    end
end

% ---------------------------------------------------------------------
function layers = buildHybridTransformerLSTM(lstmUnits, numHeads, numFeatures, l2reg)
    % Build the hybrid Transformer-LSTM model.
    % This uses MATLAB's built-in layers and a custom layer for LSTM-augmented
attention.
    % Since MATLAB does not have a direct "Transformer encoder with LSTM value"
layer,
```

```
% we simulate it using a combination of LSTM, self-attention, and addition.
% For simplicity, we use a standard self-attention layer and feed the input through an LSTM
% before attention? Actually we need to replace values with LSTM output.
% A more accurate implementation would require a custom layer. Here we propose a simplified
% architecture: LSTM layer to capture local dependencies, then multi-head self-attention,
% followed by feed-forward, with residual connections.
% This is a workaround; for exact implementation, consider using Python.

% Input layer
inputLayer = sequenceInputLayer(numFeatures, 'Normalization', 'none', 'Name', 'input');

% First LSTM to capture local patterns (outputs same sequence length)
lstmLayer1 = lstmLayer(lstmUnits, 'OutputMode', 'sequence', 'Name', 'lstm1');

% Multi-head self-attention (requires Deep Learning Toolbox R2021b or later)
% Note: numHeads must divide the feature dimension (lstmUnits). If not, adjust.
attnLayer = selfAttentionLayer(numHeads, 'Name', 'selfAttention');

% Add & Norm: addition and layer normalization
addLayer1 = additionLayer(2, 'Name', 'add1');
layernorm1 = layerNormalizationLayer('Name', 'layernorm1');

% Feed-forward network (position-wise)
ffHidden = fullyConnectedLayer(4*lstmUnits, 'Name', 'ff_hidden', 'WeightL2Factor', l2reg);
relu = reluLayer('Name', 'relu');
ffOutput = fullyConnectedLayer(lstmUnits, 'Name', 'ff_output', 'WeightL2Factor', l2reg);

% Add & Norm after FFN
addLayer2 = additionLayer(2, 'Name', 'add2');
layernorm2 = layerNormalizationLayer('Name', 'layernorm2');

% Output layer: last time step only -> regression
```

```
flatten = flattenLayer('Name', 'flatten'); % to ignore sequence dimension
fcOut = fullyConnectedLayer(1, 'Name', 'fc_out', 'WeightL2Factor', l2reg);
regLayer = regressionLayer('Name', 'output');

% Connect layers
% Input -> lstm1
% lstm1 -> selfAttention and also skip to add1
% selfAttention -> add1 (with lstm1 skip)
% add1 -> layernorm1 -> ff_hidden -> relu -> ff_output -> add2 (with layernorm1 skip)
% add2 -> layernorm2 -> flatten -> fcOut -> regLayer

% Build layer graph
lgraph = layerGraph();
lgraph = addLayers(lgraph, inputLayer);
lgraph = addLayers(lgraph, lstmLayer1);
lgraph = addLayers(lgraph, attnLayer);
lgraph = addLayers(lgraph, addLayer1);
lgraph = addLayers(lgraph, layernorm1);
lgraph = addLayers(lgraph, ffHidden);
lgraph = addLayers(lgraph, relu);
lgraph = addLayers(lgraph, ffOutput);
lgraph = addLayers(lgraph, addLayer2);
lgraph = addLayers(lgraph, layernorm2);
lgraph = addLayers(lgraph, flatten);
lgraph = addLayers(lgraph, fcOut);
lgraph = addLayers(lgraph, regLayer);

% Connect
lgraph = connectLayers(lgraph, 'input', 'lstm1');
lgraph = connectLayers(lgraph, 'lstm1', 'selfAttention');
lgraph = connectLayers(lgraph, 'lstm1', 'add1/in1');
lgraph = connectLayers(lgraph, 'selfAttention', 'add1/in2');
lgraph = connectLayers(lgraph, 'add1', 'layernorm1');
lgraph = connectLayers(lgraph, 'layernorm1', 'ff_hidden');
lgraph = connectLayers(lgraph, 'ff_hidden', 'relu');
```

```
    lgraph = connectLayers(lgraph, 'relu', 'ff_output');
    lgraph = connectLayers(lgraph, 'ff_output', 'add2/in1');
    lgraph = connectLayers(lgraph, 'layernorm1', 'add2/in2');
    lgraph = connectLayers(lgraph, 'add2', 'layernorm2');
    lgraph = connectLayers(lgraph, 'layernorm2', 'flatten');
    lgraph = connectLayers(lgraph, 'flatten', 'fc_out');
    lgraph = connectLayers(lgraph, 'fc_out', 'output');

    layers = lgraph;
end

% ---------------------------------------------------------------------
function [bestPos, bestFit, convCurve] = BKA_optimizer(objFunc, lb, ub, Npop, MaxIter, pa_init, pa_final, sigma_cauchy, lambda_leader, eta_elite, tau_stagnation)
    % Black-winged Kite Algorithm for hyperparameter optimization.
    % objFunc: handle to objective function returning fitness (to minimize)
    % lb, ub: lower and upper bounds for each dimension
    % Npop: population size
    % MaxIter: maximum iterations
    % pa_init, pa_final: attacking probability bounds
    % sigma_cauchy: scale for Cauchy mutation
    % lambda_leader: leadership coefficient
    % eta_elite: elite perturbation factor
    % tau_stagnation: stagnation detection threshold
    % Outputs:
    %     bestPos: best found position
    %     bestFit: best fitness value
    %     convCurve: convergence curve (best fitness per iteration)

    dim = length(lb);
    % Initialize population
    pop = lb + (ub - lb) .* rand(Npop, dim);
    fit = zeros(Npop, 1);
    for i = 1:Npop
        fit(i) = objFunc(pop(i, :));
```

```
    end
    [bestFit, bestIdx] = min(fit);
    bestPos = pop(bestIdx, :);

    convCurve = zeros(MaxIter, 1);
    stagnCount = 0;
    prevBest = bestFit;

    for iter = 1:MaxIter
        % Update attacking probability linearly
        pa = pa_init - (pa_init - pa_final) * (iter / MaxIter);

        for i = 1:Npop
            % Attacking behavior (exploitation)
            if rand() < pa
                r1 = rand(1, dim);
                r2 = rand(1, dim);
                % Randomly select another individual
                randIdx = randi(Npop);
                while randIdx == i
                    randIdx = randi(Npop);
                end
                newPos = pop(i, :) + r1 .* (bestPos - pop(i, :)) + r2 .* (pop(randIdx, :) -
pop(i, :));
            else
                % Soaring behavior with Cauchy mutation (exploration)
                cauchyRand = sigma_cauchy * tan(pi * (rand(1, dim) - 0.5)); % Cauchy
distribution
                newPos = bestPos + cauchyRand .* (bestPos - pop(i, :));
            end

            % Boundary handling (clip to bounds)
            newPos = max(min(newPos, ub), lb);

            % Evaluate new position
            newFit = objFunc(newPos);
```

```
        % Greedy selection
        if newFit < fit(i)
            pop(i, :) = newPos;
            fit(i) = newFit;
        end

        % Update global best
        if newFit < bestFit
            bestFit = newFit;
            bestPos = newPos;
        end
    end

    % Leadership mechanism: elite perturbation if stagnation
    if bestFit < prevBest
        stagnCount = 0;
        prevBest = bestFit;
    else
        stagnCount = stagnCount + 1;
    end

    if stagnCount >= tau_stagnation
        % Perturb the best solution
        eliteNew = bestPos + eta_elite * (ub - lb) .* randn(1, dim);
        eliteNew = max(min(eliteNew, ub), lb);
        eliteFit = objFunc(eliteNew);
        if eliteFit < bestFit
            bestFit = eliteFit;
            bestPos = eliteNew;
        end
        stagnCount = 0; % reset
    end

    convCurve(iter) = bestFit;
```

```
            % fprintf('Iter %d: bestFit = %f\n', iter, bestFit);
        end
    end

    % ---------------------------------------------------------------------
    function model = trainHybridModel(X_train, y_train, X_val, y_val, params)
        % Train the final model with optimal parameters and return trained network.
        learningRate = params(1);
        lstmUnits = round(params(2));
        numHeads = round(params(3));
        l2reg = params(4);

        layers = buildHybridTransformerLSTM(lstmUnits, numHeads, size(X_train{1},2),
    l2reg);

        options = trainingOptions('adam', ...
            'InitialLearnRate', learningRate, ...
            'MaxEpochs', 200, ...
            'MiniBatchSize', 32, ...
            'ValidationData', {X_val, y_val}, ...
            'ValidationFrequency', 50, ...
            'Verbose', true, ...
            'Plots', 'training-progress');

        net = trainNetwork(X_train, y_train, layers, options);
        model.net = net;
        model.params = params;
    end

    % ---------------------------------------------------------------------
    function y_pred = predictHybridModel(model, X_test)
        y_pred = predict(model.net, X_test);
    end

    % ---------------------------------------------------------------------
```

```
function x_denorm = denormalize(x_norm, minVal, maxVal)
    x_denorm = x_norm * (maxVal - minVal) + minVal;
end

% ------------------------------------------------------------------------
function levels = fuzzyEarlyWarning(residuals, actual, varargin)
    % Fuzzy comprehensive evaluation to assign warning levels.
    % Inputs:
    %     residuals: vector of prediction residuals (e = actual - predicted)
    %     actual     : vector of actual task frequencies
    % Optional (via varargin): thresholds struct (if not provided, computed from data)
    % Output:
    %     levels: integer vector 1=Safe, 2=Mild, 3=Moderate, 4=Severe

    % Define four indicators
    absRes = abs(residuals);
    relRes = absRes ./ (actual + eps) * 100; % percent, avoid division by zero
    w = 6; % window for cumulative and volatility
    cumRes = movsum(absRes, w, 'Endpoints', 'discard'); % cumulative over last w
    % Pad to same length as input
    cumRes = [zeros(w-1,1); cumRes];
    volRes = movstd(residuals, w, 0, 'Endpoints', 'discard'); % std over w
    volRes = [zeros(w-1,1); volRes];

    % Determine thresholds (can be based on percentiles of historical data)
    % Here we use fixed thresholds as example; in practice compute from training residuals.
    thresholds.I1 = [0, 2, 5, 10]; % for absolute residual
    thresholds.I2 = [0, 5, 15, 30]; % for relative residual
    thresholds.I3 = [0, 10, 25, 50]; % for cumulative residual
    thresholds.I4 = [0, 3, 7, 12]; % for volatility

    % Compute membership degrees for each indicator to each level
    N = length(residuals);
    membership = zeros(N, 4, 4); % indicator x level x sample
    for t = 1:N
```

```
        membership(t,1,:) = trapezoidalMembership(absRes(t), thresholds.I1);
        membership(t,2,:) = trapezoidalMembership(relRes(t), thresholds.I2);
        membership(t,3,:) = trapezoidalMembership(cumRes(t), thresholds.I3);
        membership(t,4,:) = trapezoidalMembership(volRes(t), thresholds.I4);
    end

    % Entropy weight method to determine weights for indicators
    % Compute entropy for each indicator over all samples
    epsEntropy = 1e-12;
    w_weights = zeros(4,1);
    for k = 1:4
        % Sum membership over levels for each sample (degree of fuzziness)
        p = sum(squeeze(membership(:,k,:)), 2) + epsEntropy; % Nx1
        p = p / sum(p);
        entropy_k = -sum(p .* log(p)) / log(N);
        w_weights(k) = 1 - entropy_k;
    end
    w_weights = w_weights / sum(w_weights); % normalize

    % Compute comprehensive membership for each level
    compMem = zeros(N, 4);
    for t = 1:N
        for r = 1:4
            compMem(t,r) = sum(w_weights .* squeeze(membership(t,:,r)));
        end
    end

    % Assign level by max membership
    [~, levels] = max(compMem, [], 2);
end

% --------------------------------------------------------------------
function mu = trapezoidalMembership(x, thresholds)
    % Compute membership degree to four levels based on trapezoidal functions.
    % thresholds = [a1, a2, a3, a4] with a1<a2<a3<a4
```

```
% Returns mu = [mu_I, mu_II, mu_III, mu_IV]
a1 = thresholds(1); a2 = thresholds(2); a3 = thresholds(3); a4 = thresholds(4);
mu = zeros(1,4);

% Level I (Safe)
if x <= a1
    mu(1) = 1;
elseif x < a2
    mu(1) = (a2 - x) / (a2 - a1);
else
    mu(1) = 0;
end

% Level II (Mild)
if x <= a1 || x >= a3
    mu(2) = 0;
elseif x < a2
    mu(2) = (x - a1) / (a2 - a1);
elseif x < a3
    mu(2) = (a3 - x) / (a3 - a2);
end

% Level III (Moderate)
if x <= a2 || x >= a4
    mu(3) = 0;
elseif x < a3
    mu(3) = (x - a2) / (a3 - a2);
elseif x < a4
    mu(3) = (a4 - x) / (a4 - a3);
end

% Level IV (Severe)
if x <= a3
    mu(4) = 0;
elseif x < a4
```

```
        mu(4) = (x - a3) / (a4 - a3);
    else
        mu(4) = 1;
    end
end

% ---------------------------------------------------------------------
function plotEarlyWarning(actual, predicted, levels)
    % Plot actual vs predicted with background colors for warning levels.
    figure;
    t = 1:length(actual);
    plot(t, actual, 'k-', 'LineWidth', 1.5); hold on;
    plot(t, predicted, 'r--', 'LineWidth', 1.2);
    xlabel('Sample Index (day)');
    ylabel('Task Frequency (counts/hour)');
    legend('Actual', 'Predicted', 'Location', 'best');
    title('Forecast with Graded Early Warning');

    % Add colored patches for warning levels
    colors = [0 1 0; 1 1 0; 1 0.5 0; 1 0 0]; % green, yellow, orange, red
    yLim = ylim;
    for i = 1:length(levels)-1
        x = [i, i+1, i+1, i];
        y = [yLim(1), yLim(1), yLim(2), yLim(2)];
        patch(x, y, colors(levels(i),:), 'FaceAlpha', 0.2, 'EdgeColor', 'none');
    end
    hold off;
end
```