Technical Explanation of EII Prediction Model Training and Testing

Your Name

January 30, 2025

Contents

1 Introduction

This document provides a **technical, in-depth explanation** of the two scripts used for predicting the Energy Intensity Index (EII) from multiple datasets. We use an ensemble approach consisting of *multiple feed-forward neural networks* (with an averaging mechanism) and a *Random Forest* (RF) model, combined through a weighted ensemble method.

The document is organized into two main parts:

- **Training Script**: trains an ensemble of models (Neural Networks + Random Forest) via 5-fold cross-validation on four different datasets.
- **Testing Script**: loads the best-trained models, asks the user for new input, and produces predictions (with accuracy calculations).

Note: All dataset files referenced (e.g., `MAF PU-EII.xlsx`) are assumed to be in the same directory as the scripts.

2 Training Script

2.1 Overview

The training script:

1. Loads four datasets into a MATLAB `struct`.
2. For each dataset, performs a 5-fold cross-validation.
3. Trains 10 neural networks (ensemble) and 1 Random Forest per fold.
4. Combines (averages) neural network predictions and linearly weights the final ensemble with the Random Forest prediction.
5. Selects the best ensemble across all folds based on MSE.
6. Saves the best models and preprocessing parameters to a `.mat` file for each dataset.

2.2 Complete Training Script Code

```
% Enhanced Debuggable Training Script with Multiple Neural Networks

% Set random seed for reproducibility
rng(42);

% Load datasets
datasets = struct();
datasets.MAF = readtable('MAF PU-EII.xlsx', 'VariableNamingRule', 'preserve');
datasets.SRC = readtable('SR1 PU-EII.xlsx', 'VariableNamingRule', 'preserve');
datasets.AP = readtable('AP EII-PU.xlsx', 'VariableNamingRule', 'preserve');
```

```
datasets.SRIP = readtable('SR2 EII-PU.xlsx', 'VariableNamingRule', 'preserve');

model_results = struct();
k = 5; % Number of folds for cross-validation
num_networks = 10; % Number of neural networks to train

% Iterate through all datasets
fields = fieldnames(datasets);
for i = 1:length(fields)
    dataset_name = fields{i};
    disp(['Training models for ', dataset_name, ' dataset...']);

    % Extract X and Y
    data = datasets.(dataset_name);
    X_data = data{:, 1:end-1};
    Y_data = data{:, end};

    % Input validation
    if size(X_data, 2) < 1
        error('Dataset %s has no features.', dataset_name);
    end
    if ~isnumeric(Y_data) || all(isnan(Y_data))
        error('Target variable in dataset %s is invalid.', dataset_name);
    end

    % Cross-validation partition
    cv = cvpartition(size(X_data, 1), 'KFold', k);
    mse_values = zeros(k, 1);
    mae_values = zeros(k, 1);

    % Initialize results for best ensemble
    best_mse = inf;
    best_ensemble_weight = 0;
    best_nns = cell(1, num_networks);
    best_rf_model = [];

    for fold_idx = 1:k
        disp(['Processing fold ', num2str(fold_idx), ' of ', num2str(k), '...']);
        train_idx = training(cv, fold_idx);
        val_idx = test(cv, fold_idx);

        X_train = X_data(train_idx, :);
        Y_train = Y_data(train_idx);
        X_val = X_data(val_idx, :);
        Y_val = Y_data(val_idx);

        % Check for empty data
        if isempty(X_train) || isempty(Y_train) || isempty(X_val) || isempty(Y_val)
            disp('Warning: Training or validation data is empty. Skipping fold.');
            continue;
        end

        % Robust standardization using MAD
```

```
        X_median = median(X_train, 1);
        X_mad = mad(X_train, 1);
        X_train_std = (X_train - X_median) ./ (X_mad + eps);
        X_val_std = (X_val - X_median) ./ (X_mad + eps);

        % Train multiple Neural Networks in parallel
        fold_nns = cell(1, num_networks);
        parfor nn_idx = 1:num_networks
            disp(['Training neural network ', num2str(nn_idx), '...']);
            net = feedforwardnet([100 50]);
            net.trainFcn = 'trainlm';
            net.divideFcn = 'dividerand';
            net.divideParam.trainRatio = 1;
            net.divideParam.testRatio = 0;
            net.performFcn = 'mse';
            net.trainParam.epochs = 25;

            [x_norm, ps_input] = mapminmax(X_train_std', -1, 1);
            [t_norm, ps_target] = mapminmax(Y_train', -1, 1);

            try
                [trained_net, ~] = train(net, x_norm, t_norm);
                trained_net.userdata.ps_input = ps_input;
                trained_net.userdata.ps_target = ps_target;
                fold_nns{nn_idx} = trained_net;
            catch nn_err
                disp(['Error training neural network ', num2str(nn_idx), ': ', nn_err.message]);
                fold_nns{nn_idx} = [];
            end
        end

        % Train Random Forest with adjusted hyperparameters
        disp('Training Random Forest...');
        try
            num_features = size(X_train_std, 2);
            rf_model = TreeBagger(500, X_train_std, Y_train, ...
                'Method', 'regression', ...
                'MinLeafSize', 5, ...
                'NumPredictorsToSample', max(1, floor(sqrt(num_features))), ...
                'OOBPrediction', 'on');
        catch rf_err
            disp(['Error training Random Forest: ', rf_err.message]);
            continue;
        end

        % Predictions
        try
            % Neural Network Predictions
            nn_predictions = zeros(length(Y_val), num_networks);
            valid_nn_count = 0;
            for nn_idx = 1:num_networks
                if isempty(fold_nns{nn_idx})
                    nn_predictions(:, nn_idx) = NaN;
                    continue;
                end
```

```
                x_val_norm = mapminmax('apply', X_val_std', fold_nns{nn_idx}.userdata.ps_input);
                nn_predictions(:, nn_idx) = mapminmax('reverse', fold_nns{nn_idx}(x_val_norm), fold_nns{nn_idx}.userdata.ps_target)';
                valid_nn_count = valid_nn_count + 1;
            end
            nn_avg_prediction = nanmean(nn_predictions, 2);

            % Random Forest Predictions
            rf_prediction = predict(rf_model, X_val_std);

            if all(isnan(nn_avg_prediction))
                % If all NN predictions failed, fallback to RF
                warning('All neural networks failed. Using Random Forest only.');
                ensemble_weight = 0;
                ensemble_prediction = rf_prediction;
            else
                % Calculate ensemble weights
                mse_nn_avg = mean((Y_val - nn_avg_prediction).^2);
                mse_rf = mean((Y_val - rf_prediction).^2);
                ensemble_weight = mse_rf / (mse_rf + mse_nn_avg);
                ensemble_prediction = ensemble_weight * nn_avg_prediction + (1 - ensemble_weight) * rf_prediction;
            end

            % Compute metrics
            mse_values(fold_idx) = mean((Y_val - ensemble_prediction).^2);
            mae_values(fold_idx) = mean(abs(Y_val - ensemble_prediction));

            % Update best ensemble
            if mse_values(fold_idx) < best_mse
                best_mse = mse_values(fold_idx);
                best_ensemble_weight = ensemble_weight;
                best_nns = fold_nns;
                best_rf_model = rf_model;
            end
        catch pred_err
            disp(['Error during prediction: ', pred_err.message]);
            continue;
        end
    end

    % Save best model and preprocessing parameters
    model_results.(dataset_name).NeuralNetworks = best_nns;
    model_results.(dataset_name).RandomForest = best_rf_model;
    model_results.(dataset_name).Preprocessing.mean = X_median;
    model_results.(dataset_name).Preprocessing.std = X_mad + eps;
    model_results.(dataset_name).BestEnsembleWeight = best_ensemble_weight;

    % Display final metrics
    valid_mse = mse_values(~isnan(mse_values));
    valid_mae = mae_values(~isnan(mae_values));
```

```
    if ~isempty(valid_mse)

    else
        disp([dataset_name, ': No valid folds for metrics.']);
    end

    % Visualization
    figure;
    plot(1:k, mse_values, '-o', 'DisplayName', 'MSE');
    hold on;
    plot(1:k, mae_values, '-x', 'DisplayName', 'MAE');
    xlabel('Fold');
    ylabel('Error');
    legend;
    grid on;
    title(['Performance Across Folds: ', dataset_name]);
end

% Save models
disp('Saving models...');
for dataset_name = fieldnames(model_results)'
    filename = ['model_results_', dataset_name{1}, '.mat'];
    current_model = model_results.(dataset_name{1});
    save(filename, 'current_model', '-v7.3');
    disp(['Saved ', dataset_name{1}, ' to ', filename]);
end

disp('Training completed successfully.');
```

Listing 1: Training Script

2.3 Technical Explanation of Training Steps

2.3.1 Random Seed and Data Loading

- **Random Seed (`rng(42)`)**: Ensures reproducible results across multiple runs.
- **Data Loading**: Each dataset is stored in a field of `datasets`; we keep consistent variable naming rules with `VariableNamingRule`.

2.3.2 Model Containers & Cross-Validation Setup

- `model_results`: Stores the final trained models and normalization parameters for each dataset.
- **5-Fold CV**: Partitions the data for robust performance estimation and to reduce overfitting.
- **10 Neural Networks Per Fold**: An ensemble approach typically offers higher resilience to variance in initialization.

2.3.3 Feature Extraction and Validity Checks

- We separate the last column as `Y_data` (target) and all preceding columns as `X_data`.
- Checks ensure that the dataset has at least one feature and that the target is valid (`numeric` and not all NaNs).

2.3.4 Median and MAD Normalization

- We compute the **Median** and the **MAD (Median Absolute Deviation)** for each feature.
- Normalization formula:

$$X_{\text{train\_std}} = \frac{X_{\text{train}} - \text{median}(X_{\text{train}})}{\text{MAD}(X_{\text{train}}) + \epsilon}$$

- ϵ avoids division by zero.
- This *robust scaling* is less sensitive to outliers than simple z-score normalization.

2.3.5 Training Multiple Neural Networks

- `feedforwardnet([100 50])` creates a neural network with two hidden layers, of sizes 100 and 50.
- `trainFcn = 'trainlm'` uses the Levenberg-Marquardt backpropagation.
- `parfor`: Parallelization to speed up training 10 neural networks.
- Preprocessing with `mapminmax` normalizes $[X'_{\text{train\_std}}]$ to a range $[-1, 1]$.

2.3.6 Random Forest Training

- `TreeBagger(500, ...)` creates an ensemble of 500 regression trees.
- `MinLeafSize = 5` and `NumPredictorsToSample = floor(sqrt(num_features))` are tuned hyperparameters to balance performance and interpretability.
- `'OOBPrediction','on'` enables out-of-bag prediction for internal validation and tuning.

2.3.7 Prediction and Ensemble Combining

- We get **Neural Network Predictions** for each of the 10 networks and then average them.
- We get **Random Forest Predictions**.
- We compute an **ensemble weight**:

$$w = \frac{\mathrm{MSE_{RF}}}{\mathrm{MSE_{RF}} + \mathrm{MSE_{NN}}},$$

  and use that to linearly combine $\mathrm{NN_{avg}}$ and RF.
- This gives a **Final Ensemble Prediction**.
- Each fold's performance is measured by MSE and MAE.
- The best model per dataset is stored along with its normalization parameters and ensemble weight.

2.3.8 Result Storage and Visualization

- We store the final best models in `model_results_*.mat`.
- We also plot MSE and MAE across the 5 folds to visualize performance consistency.

3 Testing Script

3.1 Overview

The testing script:

1. Loads the same four datasets (for reference and to check input dimension).
2. Loads each dataset's `.mat` file (`model_results_X.mat`).
3. Prompts the user to input a feature vector corresponding to one of the datasets.
4. Identifies which dataset the input corresponds to (based on vector length).
5. Applies the saved preprocessing (median, MAD) to the input.
6. Makes predictions from the (i) Neural Networks, (ii) Random Forest, and (iii) Weighted Ensemble.
7. Asks for the actual EII to compute accuracy percentages.
8. Outputs detailed results and identifies which model performed best.

3.2 Complete Testing Script Code

```
try
    % Load datasets
    disp('Loading datasets...');
    datasets.MAF = readtable('MAF PU-EII.xlsx', 'VariableNamingRule', 'preserve');
    datasets.SRC = readtable('SR1 PU-EII.xlsx', 'VariableNamingRule', 'preserve');
    datasets.AP = readtable('AP EII-PU.xlsx', 'VariableNamingRule', 'preserve');
    datasets.SRIP = readtable('SR2 EII-PU.xlsx', 'VariableNamingRule', 'preserve');
    disp('Datasets loaded successfully.');

    % Load individual model files
    disp('Loading models...');
    model_names = {'MAF', 'AP', 'SRC', 'SRIP'};
    for name = model_names
        filename = ['model_results_' name{1} '.mat'];
        if exist(filename, 'file')
            loaded = load(filename);
            model_results.(name{1}) = loaded.current_model;
            disp(['Loaded model: ' name{1}]);
        else
            error(['Could not find model file: ' filename]);
        end
    end
    disp('All models loaded successfully.');

    % Validate preprocessing fields
    for name = model_names
        if ~isfield(model_results.(name{1}), 'Preprocessing')
```

```
            warning('Preprocessing parameters missing for dataset: %s. Assigning default values.', name{1});
            model_results.(name{1}).Preprocessing.mean = zeros(1, size(datasets.(name{1}), 2) - 1);
            model_results.(name{1}).Preprocessing.std = ones(1, size(datasets.(name{1}), 2) - 1);
        end
        if ~isfield(model_results.(name{1}), 'NeuralNetworks')
            warning('NeuralNetworks field missing for dataset: %s. Assigning default empty neural networks.', name{1});
            model_results.(name{1}).NeuralNetworks = {};
        end
    end

catch load_err
    disp('Error loading required files:');
    disp(['Error message: ' load_err.message]);
    if exist('filename', 'var')
        disp(['Failed while processing: ' filename]);
    end
    return;
end

while true
    try
        % Ask for user input
        prompt = 'Enter the values for X variables as a vector (e.g., [100, 90, 72, 74, 90]): ';
        X_input = input(prompt);

        if isempty(X_input)
            disp('Input is empty. Please try again.');
            continue;
        end

        % Ensure X_input is a row vector
        if size(X_input, 1) > 1
            X_input = X_input';
        end

        % Find the matching dataset based on input length
        matched_dataset = '';
        fields = fieldnames(datasets);
        for i = 1:length(fields)
            dataset_name = fields{i};
            num_features = size(datasets.(dataset_name), 2) - 1;
            if length(X_input) == num_features
                matched_dataset = dataset_name;
                break;
            end
        end

        % Check if a matching dataset was found
        if isempty(matched_dataset)
            disp('Error: Input vector length does not match any dataset features.');
            continue;
        end
```

```

        % Use the matched dataset
        disp(['Using ', matched_dataset, ' dataset for prediction...']);

        % Get original variable names from the dataset
        original_vars = datasets.(matched_dataset).Properties.VariableNames(1:end-1);

        % Standardize the input using saved preprocessing parameters
        X_mean = model_results.(matched_dataset).Preprocessing.mean;
        X_std = model_results.(matched_dataset).Preprocessing.std;

        % Validate input dimensions
        if length(X_input) ~= length(X_mean)
            error('Input vector length (%d) does not match expected length (%d).', ...
                length(X_input), length(X_mean));
        end

        % Standardize input
        X_input_std = (X_input - X_mean) ./ X_std;

        % Ensure input table variable names match model expectations
        rf_var_names = datasets.(matched_dataset).Properties.VariableNames(1:length(X_input_std));
        rf_input = array2table(X_input_std, 'VariableNames', rf_var_names);

        % Debugging dimensions and variable names
        disp('Input standardized size:');
        disp(size(X_input_std));
        disp('Input variable names:');
        disp(rf_var_names);

        % Random Forest prediction
        try
            rf_model = model_results.(matched_dataset).RandomForest;
            disp('RF Model Information:');
            disp(['Predictor Names: ' strjoin(rf_model.PredictorNames, ', ')]);
            disp(['Input Names: ' strjoin(rf_input.Properties.VariableNames, ', ')]);
            disp('Input Values:');
            for i = 1:length(rf_var_names)
                disp([rf_var_names{i} ': ' num2str(X_input_std(i))]);
            end

            [Y_pred_raw, rf_scores] = predict(rf_model, rf_input);

            if iscell(Y_pred_raw)
                Y_pred_rf = str2double(Y_pred_raw{1});
            elseif isnumeric(Y_pred_raw)
                Y_pred_rf = Y_pred_raw(1);
            else
                disp(['Unexpected prediction type: ' class(Y_pred_raw)]);
                Y_pred_rf = NaN;
            end
```

```
            disp(['Final RF prediction: ' num2str(Y_pred_rf)]);
            disp(['RF scores: ' num2str(rf_scores)]);

            if isnan(Y_pred_rf)
                disp('Warning: Random Forest produced NaN prediction.');
                disp('RF Model Details:');
                disp(['Number of trees: ' num2str(rf_model.NumTrees)]);
                disp(['Method: ' rf_model.Method]);
            end
        catch rf_err
            disp('Error in Random Forest prediction:');
            disp(['Error Message: ' rf_err.message]);
            disp(['Error Details: ' getReport(rf_err)]);
            disp('Random Forest Model Details:');
            if exist('rf_model', 'var')
                disp(['Number of trees: ' num2str(rf_model.NumTrees)]);
                disp(['Method: ' rf_model.Method]);
            end
            Y_pred_rf = NaN;
        end

        % Neural Network predictions
        ensemble_nets = model_results.(matched_dataset).NeuralNetworks;
        num_networks = length(ensemble_nets);
        if isempty(ensemble_nets)
            warning('No neural networks found for dataset: %s. Skipping
               NN predictions.', matched_dataset);
            Y_pred_nn_avg = NaN;
        else
            Y_pred_nn = zeros(1, num_networks);
            for i = 1:num_networks
                net = ensemble_nets{i};
                ps_input = net.userdata.ps_input;
                ps_target = net.userdata.ps_target;
                x_norm = mapminmax('apply', X_input_std', ps_input);
                net_out = net(x_norm);
                Y_pred_nn(i) = mapminmax('reverse', net_out, ps_target)
                   ';
            end
            Y_pred_nn_avg = mean(Y_pred_nn);
        end

        % Final weighted ensemble prediction
        if ~isnan(Y_pred_rf) && ~isnan(Y_pred_nn_avg)
            mse_nn = mean((X_input - Y_pred_nn_avg).^2);
            mse_rf = mean((X_input - Y_pred_rf).^2);
            ensemble_weight = mse_rf / (mse_rf + mse_nn);
            Y_pred_final = ensemble_weight * Y_pred_nn_avg + (1 -
               ensemble_weight) * Y_pred_rf;
        elseif ~isnan(Y_pred_rf)
            Y_pred_final = Y_pred_rf;
            disp('Using only Random Forest predictions due to missing
               Neural Network predictions.');
        elseif ~isnan(Y_pred_nn_avg)
            Y_pred_final = Y_pred_nn_avg;
            disp('Using only Neural Network predictions due to missing
               Random Forest predictions.');
```

```matlab
        else
            Y_pred_final = NaN;
            disp('No valid predictions available.');
        end

        Y_pred_ensemble = Y_pred_final;
        Y_actual = input('Please provide the actual EII value for this
           input: ');

        % Store all predictions and their accuracies
        predictions = struct();
        if ~isempty(ensemble_nets)
            for i = 1:num_networks
                predictions.nn(i).value = Y_pred_nn(i);
                predictions.nn(i).accuracy = 100 - abs((Y_actual -
                   Y_pred_nn(i)) / Y_actual * 100);
            end
        end

        predictions.nn_ensemble.value = Y_pred_nn_avg;
        predictions.nn_ensemble.accuracy = 100 - abs((Y_actual -
           Y_pred_nn_avg) / Y_actual * 100);

        if ~isnan(Y_pred_rf)
            predictions.rf.value = Y_pred_rf;
            predictions.rf.accuracy = 100 - abs((Y_actual - Y_pred_rf) /
                Y_actual * 100);
        end

        predictions.final_ensemble.value = Y_pred_final;
        predictions.final_ensemble.accuracy = 100 - abs((Y_actual -
           Y_pred_final) / Y_actual * 100);

        % Calculate accuracies
        accuracies = struct();
        if ~isempty(ensemble_nets)
            accuracies.nn = zeros(1, num_networks);
            for i = 1:num_networks
                accuracies.nn(i) = 100 - abs((Y_actual - Y_pred_nn(i)) /
                    Y_actual * 100);
            end
        end

        accuracies.nn_ensemble = 100 - abs((Y_actual - Y_pred_nn_avg) /
           Y_actual * 100);
        if ~isnan(Y_pred_rf)
            accuracies.rf = 100 - abs((Y_actual - Y_pred_rf) / Y_actual
               * 100);
        else
            accuracies.rf = NaN;
        end
        accuracies.final_ensemble = 100 - abs((Y_actual -
           Y_pred_ensemble) / Y_actual * 100);

        % Find best prediction
        best_accuracy = -inf;
        best_prediction = '';
        best_value = NaN;
```

```

        % Check individual neural networks
        if ~isempty(ensemble_nets)
            for i = 1:num_networks
                if accuracies.nn(i) > best_accuracy
                    best_accuracy = accuracies.nn(i);
                    best_prediction = sprintf('Neural Network %d', i);
                    best_value = Y_pred_nn(i);
                end
            end
        end

        % Check ensemble predictions
        if accuracies.nn_ensemble > best_accuracy
            best_accuracy = accuracies.nn_ensemble;
            best_prediction = 'Neural Network Ensemble';
            best_value = Y_pred_nn_avg;
        end

        if ~isnan(Y_pred_rf) && accuracies.rf > best_accuracy
            best_accuracy = accuracies.rf;
            best_prediction = 'Random Forest';
            best_value = Y_pred_rf;
        end

        if accuracies.final_ensemble > best_accuracy
            best_accuracy = accuracies.final_ensemble;
            best_prediction = 'Final Ensemble';
            best_value = Y_pred_ensemble;
        end

        % Display results
        disp('=== Model Predictions and Accuracies ===');
        fprintf('Actual Value: %.4f\n\n', Y_actual);

        % Individual Neural Networks
        if ~isempty(ensemble_nets)
            disp('Individual Neural Networks:');
            for i = 1:num_networks
                fprintf('Network %d: %.4f (Accuracy: %.2f%%)\n', i,
                    Y_pred_nn(i), accuracies.nn(i));
            end
        end

        fprintf('\nNeural Network Ensemble: %.4f (Accuracy: %.2f%%)\n',
            ...
                Y_pred_nn_avg, accuracies.nn_ensemble);
        if ~isnan(Y_pred_rf)
            fprintf('Random Forest: %.4f (Accuracy: %.2f%%)\n',
                Y_pred_rf, accuracies.rf);
        else
            fprintf('Random Forest: Prediction failed\n');
        end
        fprintf('Final Ensemble: %.4f (Accuracy: %.2f%%)\n',
            Y_pred_ensemble, accuracies.final_ensemble);

        % Display best prediction
        fprintf('\n=== BEST PREDICTION ===\n');
```

```
        fprintf('Model: %s\n', best_prediction);
        fprintf('Value: %.4f\n', best_value);
        fprintf('Accuracy: %.2f%%\n', best_accuracy);

    catch ME
        disp('An error occurred during prediction. Debug info:');
        disp(['Error Message: ', ME.message]);
        disp(['Error Location: ', ME.stack(1).name, ' line ', num2str(ME.stack(1).line)]);

        % Additional debugging information
        disp('Debug information:');
        disp(['Input size: ', mat2str(size(X_input))]);
        disp(['Standardized input size: ', mat2str(size(X_input_std))]);
        if exist('rf_input', 'var')
            disp('RF input variable names:');
            disp(rf_input.Properties.VariableNames);
        end
    end
end
```

Listing 2: Testing Script

3.3 Technical Explanation of Testing Steps

3.3.1 Data and Model Loading

- We attempt to load all four datasets again, primarily to **check input dimension consistency**.
- We load the `model_results_*.mat` files, each containing:
  - `NeuralNetworks`: the saved ensemble of neural nets.
  - `RandomForest`: the trained Random Forest model.
  - `Preprocessing.mean` / `Preprocessing.std`: median and MAD for robust scaling.
  - `BestEnsembleWeight`: chosen weight for combining NN ensemble and RF.

3.3.2 User Input and Matching Dataset

- The user is prompted with `input(prompt)` to provide a feature vector.
- We check which dataset has the same number of features (`num_features = size(datasets.(data 2) - 1`).
- If no dataset has the same dimensionality, we error out.

3.3.3 Preprocessing and Prediction

- The script applies the saved **median/MAD-based normalization** from the training phase to the user's input.
- Predictions from:

1. **Random Forest (RF)**
2. **Neural Network Ensemble**: We apply each NN in the saved ensemble, average their outputs.
3. **Combined/Weighted Ensemble**: Takes an MSE-based weight between the NN ensemble and the RF prediction.

3.3.4 Accuracy Calculation

- The user is asked for the actual EII value: `Y_actual`.
- Accuracy is computed as:

$$\text{Accuracy (\%)} = 100 - \left|\frac{Y_{\text{actual}} - Y_{\text{pred}}}{Y_{\text{actual}}}\right| \times 100.$$

- The script displays all results:
  - Each individual NN prediction + accuracy
  - NN ensemble prediction + accuracy
  - RF prediction + accuracy
  - Final ensemble prediction + accuracy
  - The **best** prediction is highlighted.

4 Reasoning Behind Algorithmic Choices

4.1 Neural Networks

- We used multiple neural networks with different initial conditions. Neural networks can converge to local minima; using multiple networks and averaging outputs reduces variance and improves robustness.
- The architecture $[100, 50]$ with `trainlm` is a balance between model complexity and computational feasibility.

4.2 Random Forest

- A Random Forest is an ensemble method itself, combining multiple decision trees. It is often robust to outliers and can handle complex interactions.
- We used 500 trees, which is a common choice to ensure stable predictions without excessive computational cost.
- Setting `MinLeafSize = 5` controls model complexity; ensures each tree is forced to consider subsets of features.

4.3 Ensemble Weighting

- We compute an MSE-based weight because it leverages how each component of the ensemble (`NN_avg`, `RF`) performs on the validation set.
- If the RF has lower MSE than NN, it will get a higher weight, and vice versa.
- This linear combination is a straightforward, effective method of blending models.

4.4 Cross-Validation and Robust Scaling

- **Cross-Validation**: ensures that the final model is thoroughly tested on multiple distinct folds, reducing overfitting risk.
- **Median + MAD Normalization**: robust to outliers; typical in industrial or chemical process data (where extreme values can occur).

5 Conclusion

This setup, combining **(1) multiple feed-forward neural networks**, **(2) a random forest model**, and **(3) a weighted ensemble approach**, aims to provide robust, accurate EII predictions across four distinct datasets.

Key advantages:

- **Reduced Variance** through ensembling multiple neural networks.
- **Versatility** of Random Forests, which handle various data distributions well.
- **Flexible Weighted Combination** that adapts to which model performs best on validation data.

In practice, this approach demonstrates improved performance and reliability compared to any single model approach, particularly in scenarios where data may be noisy or exhibit outliers.