LSTM and Conformal Prediction

Individual project report submitted in partial fulfillment for the degree of

MSc in Data Analytics

Author:
Bharath Shakthivel - 20057027

April 17th, 2025

Table of Contents

1. **Introduction**

This report details all processes taken to create a LSTM model using Julia. This is a hybrid approach of combining LSTM with conformal prediction and benchmarking the results with NAB(Numenta Anomaly Benchmark).

1.1 LSTM (Long-Short Term Memory) for Anomaly Detection

LSTM networks, a specialized type of Recurrent Neural Network (RNN), are designed to learn long-range temporal dependencies in sequential data by mitigating the vanishing gradient problem through memory cells and gated mechanisms. Originally introduced to address limitations of standard RNNs, LSTMs effectively retain relevant information over long sequences, making them well-suited for time-series modeling.

In the context of anomaly detection, LSTMs are widely used to learn normal patterns in temporal data. By forecasting future values based on learned sequences, LSTM models can identify anomalies as deviations between predicted and actual observations. This predictive capability enables detection of unexpected events in domains like traffic monitoring, finance, and system health diagnostics.

The aim is to create a model that can efficiently and consistently evaluate time series from the chosen dataset. It will be achieved by creating a robust LSTM architecture and implementing conformal prediction on top of it.

1.2 Why Julia?

Julia is chosen for this project due to its high-performance capabilities and native support for numerical computing, which makes it ideal for building deep learning models and implementing statistical methods like conformal prediction. Unlike traditional languages such as Python or R, Julia offers the speed of C while maintaining an easy-to-read syntax, making it well-suited for time-sensitive tasks like real-time anomaly detection.

Key Julia libraries such as **Flux.jl** (for LSTM modeling), **ConformalPrediction.jl** (for uncertainty quantification), and **MLJ.jl** (for model evaluation) provide a seamless ecosystem for end-to-end development. Additionally, Julia's compatibility with scientific computing workflows enables efficient experimentation, visualization, and deployment in one unified environment.

This makes Julia a powerful and pragmatic choice for implementing LSTM-based conformal anomaly detection pipelines on real-world datasets like NAB.

1.3 Conformal Prediction

Conformal prediction is a statistical framework that provides reliable, distribution-free confidence measures for model predictions, making it particularly valuable in settings where uncertainty quantification is critical. While it has been widely applied to independent and identically distributed (i.i.d.) data, extending conformal prediction to time-series data poses unique challenges due to the inherent temporal dependencies and non-exchangeable nature of sequential observations. In response to these limitations, Xu and Xie (2023), in their paper *"Conformal Prediction for Time Series,"* introduce EnbPI, an ensemble-based conformal prediction method specifically tailored for time-series applications. EnbPI addresses the breakdown of exchangeability by using bootstrapped ensembles to estimate leave-one-out residuals efficiently, avoiding the need for retraining or data splitting. The method offers formal guarantees on both marginal and conditional coverage while remaining computationally scalable and adaptive to streaming data. Demonstrated across real-world tasks such as solar energy forecasting and traffic anomaly detection, EnbPI proves to be a robust and effective tool for uncertainty quantification in dynamic, non-stationary time-series environments.

1.4 Numenta Anomaly Benchmark (NAB) Dataset

The **Numenta Anomaly Benchmark (NAB)** dataset is a comprehensive time-series corpus specifically curated for evaluating the performance of anomaly detection algorithms. It includes both real-world and synthetic datasets, all consisting of **ordered, timestamped, single-valued metrics**, with labeled anomalies in most files.

Dataset Categories

- **Real-World Data**:
    - realAWSCloudwatch: Server metrics from AWS Cloudwatch (e.g., CPU usage, network I/O).
    - realAdExchange: Online advertising data with metrics like CPC and CPM.
    - realKnownCause: Time-series data with known anomaly causes (e.g., system failures, temperature spikes).
    - realTraffic: Real-time traffic sensor data from the Twin Cities metro area.
    - realTweets: Twitter mention counts for large public companies.

- **Artificial Data**:
    - artificialNoAnomaly: Synthetic time-series with no anomalies.
    - artificialWithAnomaly: Synthetic data with injected anomalies of various types.

NAB is especially valuable for anomaly detection research due to its variety of domains (IT systems, industrial equipment, transportation, social media), clearly labeled anomalies, and a standardized evaluation framework. Hence using this real life time series data would help us build a strong foundation on knowledge about the LSTM model using Julia and combing conformal prediction.

1.5 Libraries and Packages Utilized

The implementation of the LSTM-based conformal anomaly detection model in Julia leverages a suite of specialized libraries that facilitate deep learning, data manipulation, statistical analysis, and visualization:

- **Flux.jl**: Serves as the core deep learning framework used to construct and train the LSTM architecture due to its flexibility and seamless integration with Julia's numerical ecosystem.

- **CUDA.jl**: Enables GPU acceleration to significantly enhance computational efficiency during model training and inference.

- **CSV.jl** and **DataFrames.jl**: Employed for reading, structuring, and preprocessing the NAB time-series datasets, which are provided in CSV format.

- **Dates.jl**: Used for parsing and handling timestamped data, which is essential for temporal analysis and alignment of observations.

- **FFTW.jl**: Facilitates fast Fourier transform operations, which may be optionally applied during signal preprocessing or noise reduction phases.

- **StatsBase.jl** and **Statistics.jl**: Provide statistical tools and summary metrics essential for descriptive analysis, normalization, and evaluation of prediction outputs.

- **Random.jl**: Ensures reproducibility through controlled random number generation, particularly during weight initialization and data shuffling.

- **Plots.jl**: Supports the visualization of raw time-series data, model predictions, and identified anomalies, aiding in interpretability and reporting.

- **BSON.jl**: Used for serializing and persisting trained model parameters, allowing for model checkpointing and reuse.

- **JSON.jl**: Handles the export of results and configurations in a structured format suitable for downstream processing or API communication.

- **MLJTuning.jl**: Integrated to facilitate systematic hyperparameter tuning and model evaluation, thereby enhancing the robustness and generalizability of the anomaly detection pipeline.

This combination of packages provides a comprehensive and efficient environment for developing, training, and evaluating deep learning-based time-series anomaly detection models within the Julia ecosystem.

1.6 Evaluation Approach

We evaluated our LSTM-based conformal anomaly detection model using the Numenta Anomaly Benchmark (NAB) framework, which provides a standardized methodology for assessing anomaly detection performance on time-series data. The NAB scoring system rewards early and accurate detection while penalizing delayed or false alarms. Evaluation metrics include the NAB Score, a weighted aggregate that accounts for true positives, false positives, and false negatives based on application profiles (e.g., standard, reward low FP, or reward low FN). This approach enables a robust and interpretable comparison of our model's anomaly scores against ground truth events across multiple real-world scenarios.

1.7 Model Deployment

After achieving the desired performance, deploy the final LSTM model for real-world time series data. Make sure the model is robust and capable of handling unseen data effectively.

Some keys are; Model generalizability - to confirm that the model performs well on unseen data as well as scalability to consider how the model will manage large volumes of data. These steps helped us deploy a robot and scalable LSTM model combining a conformal prediction approach that can effectively identify anomalies in real-world scenarios.

2. Literature Review

2.1 Real-World Example (LSTM for Time-Series Anomaly Detection)

The study by Malhotra et al. (2015), *“LSTM-based Encoder-Decoder for Multi-sensor Anomaly Detection,”* presents a novel application of Long Short-Term Memory (LSTM) networks for detecting anomalies in multivariate time-series data from sensors.

Key Challenges
Sequential Dependencies: Anomaly detection in sensor data requires models that can capture temporal dependencies and long-range patterns, which traditional methods often fail to model effectively.

Unsupervised Settings: Real-world applications often lack labeled anomaly data, necessitating unsupervised or semi-supervised approaches that can learn normal behavior and detect deviations.

Variable Length Sequences: Sensor sequences can vary in length, posing a challenge for fixed-size input models.

Tools and Models
LSTM Encoder-Decoder: The authors propose an unsupervised LSTM-based encoder-decoder framework that learns to reconstruct normal sequences. Anomalies are identified based on reconstruction error, which is typically higher for anomalous sequences.

Sliding Window Approach: The model processes sequences using a sliding window, allowing it to adapt to changes in temporal patterns and detect both point anomalies and structural deviations.

Summary
Malhotra ’s work demonstrates the effectiveness of LSTM-based encoder-decoder architectures in capturing complex temporal patterns for anomaly detection in multivariate time-series. It highlights the utility of deep learning models in unsupervised settings and sets the foundation for further advancements in time-series anomaly detection frameworks.

2.2 Conformal Prediction

A foundational resource for understanding conformal prediction is the book *"Algorithmic Learning in a Random World"* by Vovk, Gammerman, and Shafer, which presents a rigorous mathematical framework for this predictive approach. The authors introduce conformal prediction as a method that complements traditional machine learning models by offering valid confidence measures for individual predictions, grounded in the theory of algorithmic randomness.

This work is particularly notable for its clear distinction between *inductive* and *transductive* conformal predictors, providing theoretical justifications and practical algorithms for both. By leveraging past data to generate prediction sets with guaranteed error rates, the method enhances model reliability — a crucial factor in high-stakes domains such as anomaly detection, medical diagnostics, and finance.

The text offers a comprehensive explanation of nonconformity measures and their role in quantifying how unusual a new example is, relative to a training set. This aligns directly with our project, where conformal scores are used to identify anomalies in time-series data by measuring deviations from learned normal behavior.

2.3 Numenta Anomaly Benchmark – NAB

The Numenta Anomaly Benchmark (NAB) is a comprehensive framework developed to evaluate real-time anomaly detection algorithms on streaming time-series data, reflecting the practical constraints and challenges of deployment in dynamic environments.

Key Challenges
Real-Time Evaluation: Most anomaly detection models are not evaluated under real-time conditions. NAB addresses this by enforcing streaming constraints, where decisions must be made sequentially as data arrives.

Ambiguous Anomaly Boundaries: In real-world applications, the exact timing and extent of anomalies are often uncertain. NAB incorporates this uncertainty by scoring based on the timeliness of detection rather than strict match to fixed labels.

Benchmarking Across Domains: Time-series data can differ significantly across industries (e.g., finance, traffic, server metrics). NAB includes datasets from diverse domains, making it a robust benchmark for general-purpose anomaly detection.

Tools and Models
Labeled Datasets with Context Windows: NAB includes labeled datasets where anomalies are annotated with temporal windows rather than single points, rewarding early detection.

Scoring Profiles: NAB offers multiple scoring profiles (standard, reward low false positives, reward low false negatives), which allow practitioners to align evaluation with their specific application requirements.

Anomaly Windows and Penalization: Detections outside the anomaly window are penalized, and false positives are tracked, encouraging models that are both accurate and precise.

Summary
The NAB framework offers a realistic and domain-diverse benchmark for evaluating anomaly detection algorithms under streaming conditions. Its nuanced scoring system and emphasis on timeliness make it particularly well-suited for assessing models in applications where early and accurate anomaly detection is critical.

3. Methodology

This study presents a robust pipeline for time-series anomaly detection using a hybrid approach that combines Long Short-Term Memory (LSTM) neural networks with conformal prediction to generate prediction intervals and identify anomalous data points. The entire framework is implemented in Julia, utilizing high-performance packages such as Flux.jl for deep learning, CSV.jl and DataFrames.jl for data handling, and Plots.jl for visualization. The NAB (Numenta Anomaly Benchmark) dataset is used to evaluate the performance of the proposed model.

3.1 Data Preparation and Preprocessing

The pipeline begins by cloning the NAB repository, which contains labeled time-series datasets from various real-world domains. Each dataset is read, parsed for timestamps, and sorted chronologically. A custom timestamp parsing function handles multiple datetime formats to ensure compatibility across diverse datasets.

To enhance model adaptability, an adaptive preprocessing module dynamically selects the LSTM input sequence length (seq_len) based on median data frequency and data characteristics such as variance and anomaly density. This data-driven window size adjustment allows the model to handle varying temporal resolutions, which is particularly important for real-world anomaly detection tasks.

The dataset is then split into training and test sets, followed by normalization using the training statistics (mean and standard deviation). This ensures that the model does not learn any information from future data points, maintaining the temporal integrity of the time-series.

3.2 Sequence Generation

Normalized data is transformed into sequences using a sliding window approach. The input features (X) are sequences of length seq_len, and the targets (Y) are the next immediate values in the series. This setup aligns with the supervised learning paradigm where the LSTM is trained to predict the next value given a historical window of previous values.

3.3 LSTM Model Architecture

The core predictive model is a multi-layer LSTM network defined using Flux.jl. The architecture includes:

A single LSTM layer with 64 hidden units for capturing temporal dependencies,

A fully connected Dense layer with ReLU activation to introduce non-linearity,

Dropout for regularization to prevent overfitting,

A final Dense layer for single-point regression output.

The model is trained using the Adam optimizer with mean absolute error (MAE) as the loss function. Early stopping is incorporated to halt training when the validation loss ceases to improve beyond a minimum threshold (min_delta) for a pre-defined number of epochs (patience), thus avoiding overfitting.

3.4 Conformal Prediction and Calibration

Following training, the model's performance is enhanced using conformal prediction to generate statistically valid prediction intervals. A ConformalPredictor structure is defined to store the trained model, calibration errors, normalization parameters, and sequence length. Using the validation set (referred to here as the calibration set), absolute residual errors between predictions and true values are computed and sorted.

During inference, the model produces point predictions and attaches a prediction interval around each output using a quantile of the calibration residuals, providing robust uncertainty estimates. This allows for distribution-free anomaly detection, as points lying outside the prediction intervals are flagged as anomalous.

3.5 Anomaly Detection and Scoring

Anomalies are identified by comparing actual values with prediction intervals. If an observed value falls outside the calculated lower or upper bound, it is considered an anomaly. Furthermore, an anomaly score is computed based on the normalized distance of the observed value from the predicted interval, allowing for prioritization and interpretability of detections.

The results are formatted in accordance with the NAB benchmark scoring framework, which evaluates models based on their ability to detect anomalies early and accurately. The output includes timestamps, predicted values, prediction bounds, and anomaly scores, and is exported to CSV format for evaluation.

3.6 Visualization and Benchmarking

Visualizations are generated to depict:

The time-series data with predicted values and 90% prediction intervals, highlighting anomalies.

The normalized anomaly scores over time, with high-score anomalies visually emphasized.

The function run_nab_benchmark() orchestrates the entire pipeline, iterating over selected datasets from the NAB benchmark (e.g., realAWSCloudwatch, realTraffic, realTweets), and automatically performs training, calibration, inference, anomaly detection, and visualization for each dataset.

3.7 Evaluation metrics

To evaluate the effectiveness of the proposed LSTM-based conformal anomaly detection framework, we adopted the standardized benchmarking methodology provided by the Numenta Anomaly Benchmark (NAB). NAB offers a domain-agnostic, real-time evaluation framework for time-series anomaly detection algorithms, emphasizing both detection accuracy and timeliness. This is particularly relevant for applications where early identification of anomalous behavior is critical.

3.7.1 NAB Scoring Methodology

NAB scoring incorporates a reward-penalty system that quantifies the quality of detections based on their proximity to labeled anomaly windows. Each detection is scored positively if it occurs within an anomaly window (with higher scores for earlier detections), and negatively if it occurs outside, thereby penalizing false positives and delayed detections. The final NAB score is a weighted aggregate that accounts for true positives, false positives, and false negatives, using different profiles—such as *standard*, *reward low false positives*, and *reward low false negatives*—to simulate varying application sensitivities.

3.7.2 Evaluation Procedure

We followed Option 2 as outlined in the NAB documentation to evaluate our results prior to threshold optimization. This involved generating per-file CSV result outputs from our anomaly detection pipeline, formatted according to NAB standards. Each file contains the following fields:

- timestamp: The time of each observation.
- value: The original time-series value.
- anomaly_score: A continuous score indicating the severity of the anomaly.
- label: A binary flag (true/false) marking whether the point was predicted as an anomaly.

These result files were stored under the appropriate subdirectories within nab/results/, corresponding to the NAB dataset taxonomy (e.g., realTweets/, realAWSCloudwatch/).

To integrate our custom detector (denoted conformal_lstm), we utilized the provided NAB script:

```
python scripts/create_new_detector.py --detector conformal_lstm
```

This automatically created the required directory structure and registered the new detector in the config/thresholds.json file. We then executed the NAB evaluation suite using:

```
python run.py -d conformal_lstm --optimize --score --normalize
```

This command performs three key operations:

1. Threshold Optimization: Finds the optimal threshold for converting continuous anomaly scores into binary decisions.

2. Scoring: Computes the raw NAB score using the ground truth labels and the optimized predictions.

3. Normalization: Converts the raw score into a standardized score, enabling comparison with other detectors in the benchmark.

3.7.3 Summary

By leveraging NAB's robust evaluation methodology, our results are benchmarked in a manner that reflects real-world operational constraints. This framework not only ensures consistency across experiments but also enables direct performance comparison with other state-of-the-art anomaly detection algorithms.

4. Results

The performance of the proposed LSTM-based conformal prediction framework was evaluated on the NAB dataset using a standard train-test split strategy. After training on the training portion of each dataset, predictions and associated uncertainty intervals were generated for the test sets. Anomaly scores were computed by measuring the deviation of observed values from the predicted intervals, and results were saved in the NAB-compatible format (timestamp, value, anomaly_score, label) to facilitate benchmarking.

Although the results were successfully formatted according to NAB specifications, an issue occurred during the final execution of the NAB scoring utility, which could not be completed as expected. Consequently, the NAB official scoring script (run.py) failed to evaluate the results automatically. As a workaround, we manually compared the structure and output patterns of our results with those of other detectors within the NAB benchmark to ensure compatibility and interpret consistency in detection behavior.

To visualize performance, we plotted the time-series data alongside the predicted values and 90% conformal prediction intervals. Additionally, we highlighted detected anomalies based on a threshold of 0.5 applied to the anomaly scores. Data points exceeding this threshold were marked as anomalies and distinctly visualized in the results. These visualizations provided intuitive insight into model behavior and helped in identifying patterns missed or flagged by the model.

A quantitative comparison summarizing the average detection performance across all datasets will be included in the final results section to highlight overall detection trends and provide baseline insights into the model's generalization capabilities.

4.1 Final Results

These results are obtained from taking the average of all the dataset results and comparing with the ground truth labels.

- Average **Precision: 0.0178**
- Average **Recall: 0.2400**
- Average **F1 Score: 0.0307**

These metrics suggest that while the model is moderately capable of identifying actual anomalies when they occur (recall), it struggles with precision, often flagging normal points as anomalous.

5. Conclusion and future work

While the current implementation demonstrates promising results in generating anomaly scores and intervals using LSTM with conformal prediction, several areas remain for further exploration and improvement:

1. Integration with NAB Scorer: The most immediate priority is to resolve the technical issue encountered with the NAB scoring utility. Running the scorer will provide normalized and comparative evaluation metrics, enabling a direct performance benchmark against state-of-the-art algorithms.

2. Full Pipeline Evaluation: Extend the prediction pipeline to generate outputs for the entire NAB dataset in the required format and automate the end-to-end benchmarking process using the NAB framework.

3. Robust Model Development: While the current model is effective in detecting certain types of anomalies, it requires enhancements to handle diverse anomaly patterns across all dataset categories. A more robust architecture may include deeper LSTM layers, attention mechanisms, or hybrid models.

4. Hyperparameter Tuning: Systematic tuning of key model parameters (e.g., sequence length, hidden layer size, dropout rate, learning rate) should be conducted using

optimization frameworks such as MLJTuning.jl to improve accuracy and reduce overfitting.

5. Real-Time Adaptation and RealKnownCause Datasets: Future iterations will focus on evaluating and adapting the model for real-time anomaly detection, particularly by testing on NAB's *realKnownCause* datasets, which reflect real-world failure events with known causes. This will allow us to assess the model's applicability in critical, high-stakes environments.

6. Scalability and Generalization: Further work will explore adapting the model to streaming settings and investigating its scalability to handle high-frequency data, enabling real-world deployment.

By addressing these directions, the current framework can evolve into a more comprehensive, benchmarked, and deployable anomaly detection system suitable for a wide range of time-series applications.

6. Code

```
#To clone the repo to the Local system
url = "https://github.com/numenta/NAB.git"
run(`git clone $url`)
using Pkg
Pkg.add(["Flux", "CUDA", "CSV", "DataFrames", "Dates", "FFTW", "StatsBase",
        "Statistics", "Random", "Plots", "BSON","JSON","MLJTuning",])
using Flux, CUDA, BSON, CSV, DataFrames, Dates, Statistics, Random, Plots, JSON, BSON,
MLJTuning, FFTW, StatsBase
#using Flux: @epochs, train!
# Set random seed for reproducibility
Random.seed!(123)
# Mean Absolute Error (MAE) function
function mae(ŷ, y)
   return mean(abs.(ŷ .- y))
end

# Function to parse the Timestamp
function parse_timestamps(timestamps)
```

```
    #Common formats
    formats = [
        dateformat"yyyy-mm-dd HH:MM:SS",
        dateformat"yyyy-mm-ddTHH:MM:SS",
        dateformat"yyyy-mm-ddTHH:MM:SS.s"
    ]

    for format in formats
        try
            return DateTime.(timestamps, format)
        catch
            continue
        end
    end

    # Error Handling
    # If all formats fail, print an example timestamp and raise error
    println("Example timestamp: $(timestamps[1])")
    error("Could not parse timestamps with any of the common formats")
end


# --- Dynamic Preprocessing ---

function adaptive_preprocessing(df)
    # Convert time differences to milliseconds as numeric values
    time_diffs = diff(df.timestamp)
    time_diffs_ms = Dates.value.(time_diffs)  # Convert to integer milliseconds

    # Calculate median frequency in milliseconds
    median_freq_ms = median(time_diffs_ms)
    median_freq = Millisecond(median_freq_ms)  # Convert back to Millisecond type

    # Auto-adjust window size based on frequency
    seq_len = if median_freq < Minute(5)
        128  # High frequency
    elseif median_freq < Hour(1)
        64
    else
        32
    end

    # Handle sparse anomalies
    if :label in names(df)
        anomaly_ratio = sum(df.label) / nrow(df)
        if anomaly_ratio < 0.01
            seq_len = min(seq_len, 32)
        end
    end

    # --- Data Variance Analysis ---
    data_values = df.value
    data_var = var(data_values)

    # Variance-based scaling (absolute thresholds)
    if data_var > 1e4
        seq_len = min(seq_len + 32, 256)
    elseif data_var < 1e2
        seq_len = max(seq_len - 16, 16)
    end

    return seq_len
end


# --- Data Preparation with Train-Test Split ---
function prepare_nab_data(file_path, test_size=0.2)
    # Load NAB data
    df = CSV.File(file_path) |> DataFrame

    # Convert timestamp to DateTime
    df.timestamp = parse_timestamps(df.timestamp)

    # Sort by timestamp
    sort!(df, :timestamp)

    # Calculate split point
    n = nrow(df)
    split_idx = floor(Int, n * (1 - test_size))
    # Split data
    train_df = df[1:split_idx, :]
    test_df = df[(split_idx+1):end, :]

    # Calculate normalization parameters from training data only
    train_mean = mean(train_df.value)
    train_std = std(train_df.value)

    # Normalize both sets using training parameters
    train_df.normalized_value = (train_df.value .- train_mean) ./ train_std
    test_df.normalized_value = (test_df.value .- train_mean) ./ train_std

    # Get dynamic sequence length from training data
    seq_len = adaptive_preprocessing(train_df)

    return train_df, test_df, train_mean, train_std, split_idx, seq_len
end


# --- Prepare sequences for model training and prediction ---
function create_sequences(data::Vector{Float64}, seq_len::Int)
    num_sequences = length(data) - seq_len

    X = Matrix{Float64}(undef, seq_len, num_sequences)
    Y = Vector{Float64}(undef, num_sequences)

    for i in 1:num_sequences
        range = i:(i+seq_len-1)
        X[:, i] = data[range]
        Y[i] = data[range[end]+1]
    end

    return reshape(X, 1, seq_len, :), Y
end


# --- Train model with Early Stopping ---
function train_lstm_model(X_train, Y_train, X_val, Y_val;
    epochs=200, patience=15, min_delta=0.001)
```

```
# Define model architecture
model = Chain(
    LSTM(1 => 64),
    xs -> xs[:, end, :],
    Dense(64 => 32, relu),
    Dropout(0.2),
    Dense(32 => 1, relu),
    x -> vec(x)
)

# Optimizer
opt = Adam(0.001)
state = Flux.setup(opt, model)

# For early stopping
best_val_loss = Inf
best_model = deepcopy(model)
patience_counter = 0

# Training history
train_losses = Float32[]
val_losses = Float32[]

for epoch in 1:epochs
    # Reset LSTM state
    Flux.reset!(model)

    # Forward pass and gradient computation
    loss, grads = Flux.withgradient(model) do m
        preds = m(X_train)
        mae(preds, Y_train)
    end

    # Update model
    Flux.update!(state, model, grads)

    # Validate
    Flux.reset!(model)
    val_predictions = model(X_val)
    val_loss = mae(val_predictions, Y_val)
```

```
        # Store losses
        push!(train_losses, loss)
        push!(val_losses, val_loss)

        # Early stopping check
        if val_loss < best_val_loss - min_delta
            best_val_loss = val_loss
            best_model = deepcopy(model)
            patience_counter = 0
        else
            patience_counter += 1
            if patience_counter >= patience
                println("Early stopping at epoch $epoch")
                break
            end
        end

        # Print progress
        if epoch % 10 == 0
            println("Epoch $epoch: Train Loss = $(round(loss, digits=4)), Val Loss =
$(round(val_loss, digits=4))")
        end
    end


    # Plot training curves
    p = plot(
        train_losses,
        label="Training Loss",
        linewidth=2,
        xlabel="Epoch",
        ylabel="Loss (MAE)",
        title="Training and Validation Loss"
    )
    plot!(p, val_losses, label="Validation Loss", linewidth=2, linestyle=:dash)
    display(p)

    return best_model, train_losses, val_losses
end
```

```
# --- Conformal Prediction Implementation ---
struct ConformalPredictor
    model::Chain
    calibration_errors::Vector{Float64}
    mean::Float64
    std::Float64
    seq_len::Int64
end

function train_conformal_predictor(model, X_calib, Y_calib, mean_val, std_val, seq_len)
    # Reset LSTM state
    Flux.reset!(model)

    # Make predictions on calibration set
    predictions = model(X_calib)

    # Calculate absolute errors for calibration
    errors = abs.(predictions .- Y_calib)

    # Sort errors for quantile calculation
    sorted_errors = sort(errors)

    return ConformalPredictor(model, sorted_errors, mean_val, std_val, seq_len)
end

function predict_with_intervals(predictor::ConformalPredictor, X_new, alpha=0.1)
    # Reset LSTM state
    Flux.reset!(predictor.model)

    # Make point predictions
    point_predictions = predictor.model(X_new)

    # Get appropriate quantile from calibration errors
    n_calib = length(predictor.calibration_errors)
    quantile_idx = ceil(Int, (n_calib + 1) * (1 - alpha))

    if quantile_idx > n_calib
        quantile_idx = n_calib
    end
    # Get error margin for prediction interval
    error_margin = predictor.calibration_errors[quantile_idx]

    # Calculate prediction intervals
    lower_bound = point_predictions .- error_margin
    upper_bound = point_predictions .+ error_margin

    # Denormalize to original scale
    point_preds_original = (point_predictions .* predictor.std) .+ predictor.mean
    lower_bound_original = (lower_bound .* predictor.std) .+ predictor.mean
    upper_bound_original = (upper_bound .* predictor.std) .+ predictor.mean

    return point_preds_original, lower_bound_original, upper_bound_original
end

# --- Anomaly Detection with Conformal Prediction ---
function detect_anomalies(test_df, point_preds, lower_bound, upper_bound, seq_len)
    # Create a DataFrame for results
    results = DataFrame(
        timestamp = test_df.timestamp[seq_len+1:end],
        actual = test_df.value[seq_len+1:end],
        predicted = point_preds,
        lower_bound = lower_bound,
        upper_bound = upper_bound
    )

    # Identify anomalies (values outside the prediction interval)
    results.is_anomaly = (results.actual .< results.lower_bound) .| (results.actual .>
results.upper_bound)

    # Calculate anomaly scores (distance from prediction, normalized by interval width)
    results.anomaly_score = abs.(results.actual .- results.predicted) ./ (results.upper_bound .-
results.lower_bound)

    return results
end

# --- Convert results to NAB format ---
function convert_to_nab_format(results)
```

```
    # Format required by NAB: A csv with timestamp and anomaly likelihood
    nab_results = DataFrame(
        timestamp = string.(results.timestamp),
        value = results.actual,
        anomaly_score = results.anomaly_score,
        label = results.is_anomaly
    )

    return nab_results
end

# --- Main Execution Pipeline ---
function run_conformal_anomaly_detection(file_path)
    # Prepare data
    dataset_name = basename(file_path)
    println("Processing dataset: $dataset_name")

    # Get dynamic sequence length from data
    train_df, test_df, train_mean, train_std, split_idx, seq_len = prepare_nab_data(file_path)

    # Further split training into train and validation
    n_train = nrow(train_df)
    val_size = 0.2
    val_split_idx = floor(Int, n_train * (1 - val_size))

    train_data = train_df[1:val_split_idx, :].normalized_value
    val_data = train_df[(val_split_idx+1):end, :].normalized_value

    # Prepare sequences
    X_train, Y_train = create_sequences(train_data, seq_len)
    X_val, Y_val = create_sequences(val_data, seq_len)

    # Convert to Float32
    X_train, Y_train = Float32.(X_train), Float32.(Y_train)
    X_val, Y_val = Float32.(X_val), Float32.(Y_val)

    # Applying normalization
    x_mean = mean(X_train)
    x_std = std(X_train)
    X_train = (X_train .- x_mean) ./ x_std
    X_val = (X_val .- x_mean) ./ x_std

    y_mean = mean(Y_train)
    y_std = std(Y_train)
    Y_train = (Y_train .- y_mean) ./ y_std
    Y_val = (Y_val .- y_mean) ./ y_std

    # Train model with early stopping
    println("Training LSTM model...")
    best_model, train_losses, val_losses = train_lstm_model(X_train, Y_train, X_val, Y_val)

    # Prepare test data
    X_test, Y_test = create_sequences(test_df.normalized_value, seq_len)
    X_test, Y_test = Float32.(X_test), Float32.(Y_test)

    # Create validation+calibration set for conformal prediction
    X_calib, Y_calib = X_val, Y_val

    # Train conformal predictor
    println("Training conformal predictor...")
    conformal_predictor = train_conformal_predictor(best_model, X_calib, Y_calib, train_mean,
train_std,seq_len)

    # Make predictions with uncertainty
    println("Making predictions with uncertainty...")
    point_preds, lower_bound, upper_bound = predict_with_intervals(conformal_predictor,
X_test, 0.1)

    # Detect anomalies
    println("Detecting anomalies...")
    anomaly_results = detect_anomalies(test_df, point_preds, lower_bound, upper_bound,
seq_len)

    # Convert to NAB format
    nab_format_results = convert_to_nab_format(anomaly_results)

    # Save results for NAB scoring (NOTE: Change the path to save the results for the
corresponding data)
    # eg: "realTweets in the place of realTraffic"
    # Although we can write for loop to run as a whole but it not optimal for our use case.
    detection_file = joinpath("NAB","nab", "results", "conformal_lstm", "realAWSCloudwatch",
"conformal_lstm_$(replace(dataset_name, ".csv" => "")).csv")
    mkpath(dirname(detection_file))
    CSV.write(detection_file, nab_format_results)

    # Plot results
    plot_results(anomaly_results, dataset_name)

    # # Save model
    # save_model(conformal_predictor, dataset_name, seq_len)

    return conformal_predictor, anomaly_results, detection_file
end

# --- Visualization of results ---
function plot_results(results, dataset_name)
    # Plot the time series with prediction intervals and anomalies
    p = plot(
        results.timestamp, results.actual,
        label="Actual Values",
        linewidth=2,
        title="Anomaly Detection for $dataset_name",
        xlabel="Time",
        ylabel="Value",
        legend=:topright
    )

    plot!(p, results.timestamp, results.predicted, label="Predicted Values", linewidth=2,
linestyle=:dash)

    # Plot prediction intervals as a ribbon
    plot!(
        p,
        results.timestamp,
        results.predicted,
        ribbon=(results.predicted .- results.lower_bound, results.upper_bound .- results.predicted),
        fillalpha=0.3,
        label="90% Prediction Interval"
    )
    display(p)
    savefig(p, "anomaly_detection_$(replace(dataset_name, ".csv" => "")).png")

    # Normalize anomaly scores to [0, 1]
    normalized_scores = (results.anomaly_score .- minimum(results.anomaly_score)) ./
                     (maximum(results.anomaly_score) - minimum(results.anomaly_score))

    results[!, :normalized_score] = normalized_scores

    # Plot normalized anomaly scores
    p2 = plot(
        results.timestamp,
        results.normalized_score,
        linewidth=2,
        title="Normalized Anomaly Scores for $dataset_name",
        xlabel="Time",
        ylabel="Normalized Anomaly Score",
        legend=false
    )

    # Highlight points above threshold
    threshold = 0.5  # Adjust as needed
    high_scores = results[results.normalized_score .> threshold, :]
    scatter!(
        p2,
        high_scores.timestamp,
        high_scores.normalized_score,
        markersize=4,
        color=:red,
        label="High Anomaly Score"
    )

    display(p2)
    savefig(p2, "normalized_anomaly_scores_$(replace(dataset_name, ".csv" => "")).png")

end

# --- Complete NAB Benchmarking Pipeline ---
function run_nab_benchmark()
```

```
# Use relative paths based on NAB repo structure
datasets = [
    # # realTraffic
    # "data/realTraffic/occupancy_6005.csv",
    # "data/realTraffic/occupancy_t4013.csv",
    # "data/realTraffic/speed_6005.csv",
    # "data/realTraffic/speed_7578.csv",
    # "data/realTraffic/speed_t4013.csv",
    # "data/realTraffic/TravelTime_387.csv",
    # "data/realTraffic/TravelTime_451.csv",

    # realTweets
    #"data/realTweets/Twitter_volume_AAPL.csv"
    #"data/realTweets/Twitter_volume_AMZN.csv",
    #"data/realTweets/Twitter_volume_CRM.csv",
    #"data/realTweets/Twitter_volume_CVS.csv",
    #"data/realTweets/Twitter_volume_FB.csv",
    #"data/realTweets/Twitter_volume_GOOG.csv",
    #"data/realTweets/Twitter_volume_IBM.csv",
    #"data/realTweets/Twitter_volume_KO.csv",
    #"data/realTweets/Twitter_volume_PFE.csv",
    #"data/realTweets/Twitter_volume_UPS.csv",

    # # artificialWithAnomaly
    #"data/artificialWithAnomaly/art_daily_flatmiddle.csv",
    #"data/artificialWithAnomaly/art_daily_jumpsdown.csv",
    #"data/artificialWithAnomaly/art_daily_jumpsup.csv",
    #"data/artificialWithAnomaly/art_daily_nojump.csv",
    #"data/artificialWithAnomaly/art_increase_spike_density.csv",
    #"data/artificialWithAnomaly/art_load_balancer_spikes.csv"

    # # artificialNoAnomaly
    #"data/artificialNoAnomaly/art_flatline.csv",
    #"data/artificialNoAnomaly/art_noisy.csv",
    #"data/artificialNoAnomaly/art_daily_small_noise.csv",
    #"data/artificialNoAnomaly/art_daily_perfect_square_wave.csv",
    #"data/artificialNoAnomaly/art_daily_no_noise.csv"

    # # realKnownCause
    #"data/realKnownCause/ambient_temperature_system_failure.csv",
    #"data/realKnownCause/cpu_utilization_asg_misconfiguration.csv",
    #"data/realKnownCause/ec2_request_latency_system_failure.csv",
    #"data/realKnownCause/machine_temperature_system_failure.csv",
    #"data/realKnownCause/nyc_taxi.csv",
    #"data/realKnownCause/rogue_agent_key_hold.csv",
    #"data/realKnownCause/rogue_agent_key_updown.csv"

    # # realAdExchange
    #"data/realAdExchange/exchange-2_cpc_results.csv",
    #"data/realAdExchange/exchange-2_cpm_results.csv",
    #"data/realAdExchange/exchange-3_cpc_results.csv",
    #"data/realAdExchange/exchange-3_cpm_results.csv",
    #"data/realAdExchange/exchange-4_cpc_results.csv",
    #"data/realAdExchange/exchange-4_cpm_results.csv"

    # # realAWSCloudwatch
    "data/realAWSCloudwatch/ec2_cpu_utilization_24ae8d.csv",
    "data/realAWSCloudwatch/ec2_cpu_utilization_53ea38.csv",
    "data/realAWSCloudwatch/ec2_cpu_utilization_5f5533.csv",
    "data/realAWSCloudwatch/ec2_cpu_utilization_77c1ca.csv",
    "data/realAWSCloudwatch/ec2_cpu_utilization_825cc2.csv",
    "data/realAWSCloudwatch/ec2_cpu_utilization_ac20cd.csv",
    "data/realAWSCloudwatch/ec2_cpu_utilization_c6585a.csv",
    "data/realAWSCloudwatch/ec2_cpu_utilization_fe7f93.csv",
    "data/realAWSCloudwatch/ec2_disk_write_bytes_1ef3de.csv",
    "data/realAWSCloudwatch/ec2_disk_write_bytes_c0d644.csv",
    "data/realAWSCloudwatch/ec2_network_in_257a54.csv",
    "data/realAWSCloudwatch/ec2_network_in_5abac7.csv",
    "data/realAWSCloudwatch/elb_request_count_8c0756.csv",
    "data/realAWSCloudwatch/grok_asg_anomaly.csv",
    "data/realAWSCloudwatch/iio_us-east-1_i-a2eb1cd9_NetworkIn.csv",
    "data/realAWSCloudwatch/rds_cpu_utilization_cc0c53.csv",
    "data/realAWSCloudwatch/rds_cpu_utilization_e47b3b.csv"

]

for dataset in datasets
    println("\n=== Processing: $dataset ===")
    full_path = joinpath("NAB/", dataset)
    run_conformal_anomaly_detection(full_path)
    end
end

# Training phase (automatically determines seq_len)
# Main Function to initiate the pipeline
run_nab_benchmark()


#-------- Comparing test data results with ground truth labels----------------

using CSV, DataFrames, JSON, Dates, Statistics


function compare_anomaly_results(csv_path::String, json_path::String, dataset_filename::String;
threshold::Float64 = 1.0)
    # Step 1: Load your predictions CSV
    result_df = CSV.read(csv_path, DataFrame)
    result_df.timestamp = DateTime.(result_df.timestamp)

    # Step 2: Load NAB ground truth JSON
    labels_dict = JSON.parsefile(json_path)
    gt_times = labels_dict[dataset_filename]
    anomaly_times = DateTime.(gt_times, dateformat"yyyy-mm-dd HH:MM:SS")

    # Step 3: Add ground truth label (1 for anomaly, 0 otherwise)
    result_df.label = [t in anomaly_times ? 1 : 0 for t in result_df.timestamp]

    # Step 4: Classify predictions based on threshold
    result_df.prediction = result_df.anomaly_score .>= threshold

    # Step 5: Evaluation metrics
    TP = sum((result_df.prediction .== 1) .& (result_df.label .== 1))
    FP = sum((result_df.prediction .== 1) .& (result_df.label .== 0))
    FN = sum((result_df.prediction .== 0) .& (result_df.label .== 1))
    TN = sum((result_df.prediction .== 0) .& (result_df.label .== 0))

    precision = TP / (TP + FP + eps())
    recall = TP / (TP + FN + eps())
    f1 = 2 * (precision * recall) / (precision + recall + eps())

    println("Results for: $dataset_filename")
    println("TP: $TP, FP: $FP, FN: $FN, TN: $TN")
    println("Precision: ", round(precision, digits=4))
    println("Recall: ", round(recall, digits=4))
    println("F1 Score: ", round(f1, digits=4))

    return result_df
end

# Load your detection results
# csv_path = "C:/Users/bhara/Desktop/Output/Final Results/Results/realAWSCloudwatch/conformal_lstm_ec2_cpu_utilization_53ea38.csv"
# csv_path = "C:/Users/bhara/Desktop/Output/Final Results/Results/realAWSCloudwatch/conformal_lstm_ec2_cpu_utilization_24ae8d.csv"
# csv_path = "C:/Users/bhara/Desktop/Output/Final Results/Results/realAWSCloudwatch/conformal_lstm_ec2_cpu_utilization_77c1ca.csv"
# csv_path = "C:/Users/bhara/Desktop/Output/Final Results/Results/realAWSCloudwatch/conformal_lstm_ec2_cpu_utilization_825cc2.csv"
# csv_path = "C:/Users/bhara/Desktop/Output/Final Results/Results/realAWSCloudwatch/conformal_lstm_ec2_cpu_utilization_ac20cd.csv"
# csv_path = "C:/Users/bhara/Desktop/Output/Final
Results/Results/realAWSCloudwatch/conformal_lstm_ec2_cpu_utilization_c6585a.csv"

# csv_path = "C:/Users/bhara/Desktop/Output/Final
Results/Results/realAWSCloudwatch/conformal_lstm_ec2_cpu_utilization_fe7f93.csv"

# csv_path = "C:/Users/bhara/Desktop/Output/Final
Results/Results/realAWSCloudwatch/conformal_lstm_ec2_disk_write_bytes_1ef3de.csv"

# csv_path = "C:/Users/bhara/Desktop/Output/Final
Results/Results/realAWSCloudwatch/conformal_lstm_ec2_disk_write_bytes_c0d644.csv"

# csv_path = "C:/Users/bhara/Desktop/Output/Final
Results/Results/realAWSCloudwatch/conformal_lstm_ec2_network_in_5abac7.csv"

# csv_path = "C:/Users/bhara/Desktop/Output/Final
Results/Results/realAWSCloudwatch/conformal_lstm_ec2_network_in_257a54.csv"

# csv_path = "C:/Users/bhara/Desktop/Output/Final
Results/Results/realAWSCloudwatch/conformal_lstm_elb_request_count_8c0756.csv"

# csv_path = "C:/Users/bhara/Desktop/Output/Final
Results/Results/realAWSCloudwatch/conformal_lstm_grok_asg_anomaly.csv"

# csv_path = "C:/Users/bhara/Desktop/Output/Final
Results/Results/realAWSCloudwatch/conformal_lstm_iio_us-east-1_i-a2eb1cd9_NetworkIn.csv"

# csv_path = "C:/Users/bhara/Desktop/Output/Final
Results/Results/realAWSCloudwatch/conformal_lstm_rds_cpu_utilization_cc0c53.csv"

csv_path = "C:/Users/bhara/Desktop/Output/Final
Results/Results/realAWSCloudwatch/conformal_lstm_rds_cpu_utilization_e47b3b.csv"

# Compare with ground truth


compare_anomaly_results(csv_path, "labels/combined_labels.json",
"realAWSCloudwatch/rds_cpu_utilization_e47b3b.csv", threshold=1.0)




# ---------------- NOTE --------------- ************* ------------------------

# Manually changed the path for csv_path variable and in the function call to generate the
results.
```

7. References

Barbareschi, M.A., Alqallaf, F., Gammerman, A. and Nouretdinov, I., 2021. *Conformal prediction for time series data*. Statistical Analysis and Data Mining: The ASA Data Science Journal, 14(6), pp. 549–565. https://doi.org/10.1002/sam.11356

Chen, X. and Xie, Y., 2023. *Conformal prediction for time series*. IEEE Transactions on Pattern Analysis and Machine Intelligence. Available at: https://ieeexplore.ieee.org/document/10121511

Hochreiter, S. and Schmidhuber, J., 1997. *Long short-term memory*. Neural Computation, 9(8), pp.1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

Malhotra, P., Vig, L., Shroff, G. and Agarwal, P., 2015. *Long short term memory networks for anomaly detection in time series*. In: Proceedings of the European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning (ESANN). Available at: https://arxiv.org/pdf/1502.04681.pdf

Malhotra, P., Ramakrishnan, A., Anand, G., Vig, L., Agarwal, P. and Shroff, G., 2016. *LSTM-based Encoder-Decoder for Multi-sensor Anomaly Detection*. arXiv preprint arXiv:1607.00148. Available at: https://arxiv.org/pdf/1607.00148.pdf

Numenta, 2015. *Numenta Anomaly Benchmark (NAB)*. Available at: https://github.com/numenta/NAB

Lavin, A. and Ahmad, S., 2015. *Evaluating real-time anomaly detection algorithms – the Numenta Anomaly Benchmark*. In: 2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA). IEEE, pp. 38–44. https://ieeexplore.ieee.org/document/7424283

Vovk, V., Gammerman, A. and Shafer, G., 2005. *Algorithmic learning in a random world*. Springer Science & Business Media.