B9DA109 Machine Learning and Pattern Recognition: CA_TWO

Conformal LSTM Classifier for Anomaly Detection on NAB

January 2025

Submitted by:
Anish Rao: 20066423

Lecturer: Devesh Jawla

Index

1. Introduction

Time series data anomaly is important in fields such as cybersecurity, finance, healthcare, IT operations, and medicine where spotting the changes in data patterns may lead to predicting failures, or attacks or finding rare events. The exact objective of this project is to achieve a Long Short-Term Memory (LSTM) based classifier along with conformal prediction for the anomaly detection on the Numenta Anomaly Benchmark (NAB) dataset. The dataset is composed of more than 50 labeled real-world and artificial time-series with the anomalies indicated. It is a benchmark that is specifically developed to check the performance of real-time anomaly detection systems in diverse, noisy, and real-world situations (Ahmad et al., 2017).

While the project's main requirement was to build a conformal LSTM model, this work goes a step further by comparing two different modeling approaches.

1. **Unsupervised anomaly detection** - the model learns patterns without the help of any labels and then, detects anomalies with the aid of the forecasting error.
2. **Supervised anomaly detection** - the model is taught with labelled data directly to classify sequences as anomalous or not.

Both models share the same main structure but the way they are trained and handle the anomalies are. In both, conformal prediction helps to avoid setting thresholds manually. The models apply a calibrated limit which is decided from the data to say what is actually an anomaly and what is not. This approach aims to improve the reliability of detections and ensuring the anomaly decisions have a statistical confidence level.

Aims and Objectives

- Design and build a Conformal LSTM-based anomaly detection system in Julia.
- Try out both the unsupervised (forecasting-based) and the supervised (classification-based) training methods.
- Use conformal prediction to set statistically valid, data-driven thresholds.
- Use Precision, Recall, and F1-score metrics to verify both methods using the NAB ground truth.
- Compare the two models to see which one is more efficient on differently organized time series.

2. Literature Review

Time series anomaly detection is a deeply researched issue with lots of academic and industry applications. Many tools and methods have been created, ranging from identifying financial fraud to tracking unusual/failures in systems. The Numenta Anomaly Benchmark (NAB**)** was created to evaluate real-time anomaly detection models on a wide range of time series having both synthetic and real-world data (Ahmad et al., 2017).

State-of-the-Art Methods

Many approaches have shown strong performance on the NAB leaderboard:

- **Hierarchical Temporal Memory (HTM):** Just like the human brain interprets sequences, HTM recognize temporal patterns and flags anything that deviates from them. It is mostly useful for streaming and continuously changing data (Cui et al., 2016).
- **Bayesian Models:** These models use probability distributions to calculate how much is the possibility of a data point. The new point is marked as an anomaly if the probability is low.
- **KNN-CAD (k-Nearest Neighbors - Contextual Anomaly Detection):** This method compares current pattern with past. A value is considered as anomaly if the current pattern is very different from what it should or could have been. It has been successfully used by NASA to detect anomalies in spacecraft telemetry data (Hundman et al., 2018).
- **Forecasting Models (ARIMA, LSTM):** These models use past values to try to predict the next. The predicted value is marked as anomaly if it differs a lot from the actual value.
- **Ensemble Methods:** These are combination of 2 or more smaller models to give better overall decisions, by reducing false positives and improving model.

Industry Tools and Real-World Applications

Large tech companies also provide anomaly detection tools:

- AWS Lookout for Metrics uses machine learning to identify unforeseen shifts in key performance indicators such as traffic or revenue. It gives root cause analysis and real-time notifications (Amazon Web Services, n.d.).
- Azure Anomaly Detector is a combination of deep learning and statistical models with natural thresholding (Microsoft, n.d.).
- Google Cloud's Vertex AI gives users the option to train personalized time series models for detecting anomalies using AutoML and custom neural networks (Google Cloud, n.d.).

Strengths and Limitations of LSTM

Long Short-Term Memory networks (LSTM) are a type of neural network that are particularly made for sequence learning. It is a good choice for detecting anomalies as they capture long-term dependencies in data better than basic statistical models.

However, LSTMs aren't perfect:

- They require a lot of data for training and take a long time to train to be able to perform decently.
- Hyperparameter configuration is important and takes long time to figure out values for best results sequence/window length, learning rate, and dropout.
- Without a probabilistic output, it's quite difficult to know how confident an LSTM is making it harder to decide if the output is actually anomaly or not.
- This is the place that conformal prediction comes in. It adds a statistical layer that tells us how different a prediction is, using the residuals, and decide a reliable threshold to make decisions.

While LSTM models are good for learning from sequential data, their best results are achieved when using ensemble methods such as conformal prediction to improve the model. Some other methods like probabilistic models or ensembles may suit better where interpretability or simplicity is more important. The choice of method narrows down to the use case, data quality, and resources at hand.

3. Methodology

I developed two different ways of implementing anomaly detector in Julia. Both the supervised (pure classification) and unsupervised (forecasting + classification) methods share same LSTM architecture and use conformal prediction for thresholding. Both differ in the way the data is loaded, split and model layers. Both models are trained over all 58 datasets

3.1 Data Preparation

Parameter Loading

Before processing each dataset file, a configuration JSON file is maintained to load custom parameters:

- `train_ratio` (supervised) / `probation_ratio` (unsupervised)
- `calib_ratio`, `confidence` - for conformal prediction
- `window_size` (sequence length)
- `epochs, learning_rate, dropout`

Each file has its own set of tuned parameters. These were decided by manual experimentation. Files were grouped roughly by file length. Datasets with similar sizes were mostly similar, so the same parameters were reused for efficiency and save time. This helped avoid overfitting and improve performance on each individual file.

Data Loading

Each dataset file is csv with two columns:

- `timestamp`
- `value`

For the supervised model, a `labels.json` file is also loaded. This has labelled anomaly windows. Each timestamp is checked against these windows to assign a binary label:

- 1 for anomaly
- 0 for normal

This label is only used in the supervised model. The unsupervised model never sees any label and learns only from the values.

Data Normalization and Splitting

All of the values are standardized using the mean and standard deviation. To avoid data leakage, I normalize only using the train portion and not the complete dataset. For deciding the data split I use the "probation period" mentioned in the NAB white paper which is supposed to be anomaly free (Ahmad et al., 2017).

- If the file has fewer than 5000 points, the probation is set to 15% of the total.
- If it has more than 5000, the probation is fixed at 750 points.

From this probation period (let's say 15%):

- The first 10% is used as the training set, and only this is used for normalization (mean and standard deviation).
- The next 5% is the calibration set used for conformal prediction.
- The remaining 85% is used for evaluation.

For unsupervised model, this ensures that both training and calibration sets have clean data without anomalies, which is important for the model to learn normal behaviour.
For the supervised model it is not 100% required for training set to be anomaly free as we need the model to also know what an anomaly looks like.

Data Sequencing

To use the data with an LSTM model, it has to be converted into sliding windows as it doesn't look at a single point. These windows help the model understand patterns.

- In the unsupervised model, each window is used to predict the next point in the series.
- In the supervised model**,** each window is labelled as either normal or anomaly based on the final point in the sequence.

The NAB white paper recommended using a window size of 10% of the set. But during testing, this was adjusted manually to see what worked best for each file. In general:

- Shorter files worked better with smaller windows.
- Longer files worked better with larger windows.

These final window sizes were saved per file in the parameter JSON file and the model automatically loaded the right ones.

3.2 Model Architecture

Both the supervised and unsupervised models use the same basic LSTM structure, with small changes to layers respectively.

Model Structure

- LSTM layer – this layer learns patterns from the input windows.
- Dropout layer - this helps to reduce overfitting (tuned for each file)
- Dense Output layer:
    - For unsupervised it outputs/predicts the next value in series.
    - For supervised, the output goes through a sigmoid to give probability of classification.

Training Setup

- Unsupervised:
    - Predict the next value in the sequence.
    - Uses Mean Squared Error (MSE) loss.
- Supervised:
    - Classify if the last point in a sequence is an anomaly.
    - Uses Logit Binary Cross-Entropy loss.

Optimizer and Tuning

- Both models use the ADAMW optimizer for efficient and stable learning.
- Dropout rate, learning rate, epochs, and window size are manually tuned per file and loaded from the parameter JSON file.
- The LSTM's hidden state is reset at the start of each epoch to keep sequences independent.

3.3 Conformal Prediction

Instead of using fixed thresholds, this project uses Conformal Prediction to set the thresholds based on calibration errors.
Once training is done, the model is run on the calibration set, and the residual is calculated for each sample.

- Unsupervised:
  `residual = |predicted_value – actual_value|`
- Supervised:
  `residual = |predicted_probability - true_label|`

These residuals show how confident the model is. Using this residual and confidence level which is different for each file, the threshold is calculated. During testing, if a prediction's error is greater than this threshold, it's flagged as an anomaly.

3.4 Evaluation

The performance for both models was calculated using Precision, Recall and F1 Score.

- Precision shows how many detected anomalies were actually correct.
- Recall shows how many true anomalies were found.
- F1 Score is the balance between precision and recall.

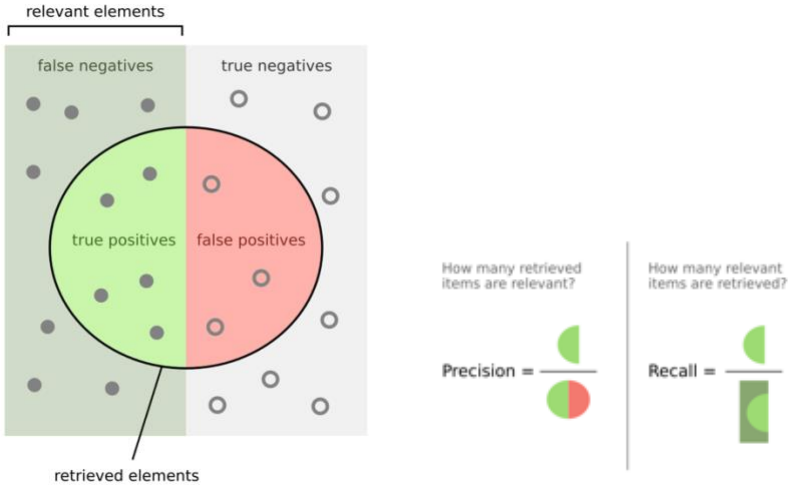

Figure 1

4. Results

The performance of both models was evaluated across all 58 files in the NAB dataset using Precision, Recall, and F1 Score. These were calculated for each file individually and then averaged across different data categories.
The table below is the sum of all true positives (TP), false positives (FP) and false negatives (FN) for both models.

Model	True Positives (Correct Detections)	False Positives (Incorrect Detections)	False Negatives (Missed Anomalies)
Supervised	8780	37991	37362
Unsupervised	16585	76082	34569

The unsupervised model detected nearly double TP's, but it also gave more than double FP's. This means that it is more sensitive.

The supervised model raised fewer false alarms but missed a lot of actual anomalies, meaning it is more relaxed.

Figure 2 shows average F1 scores for the 7 main categories in NAB dataset. Unsupervised model achieved better scores in 4 of 6 NAB categories.

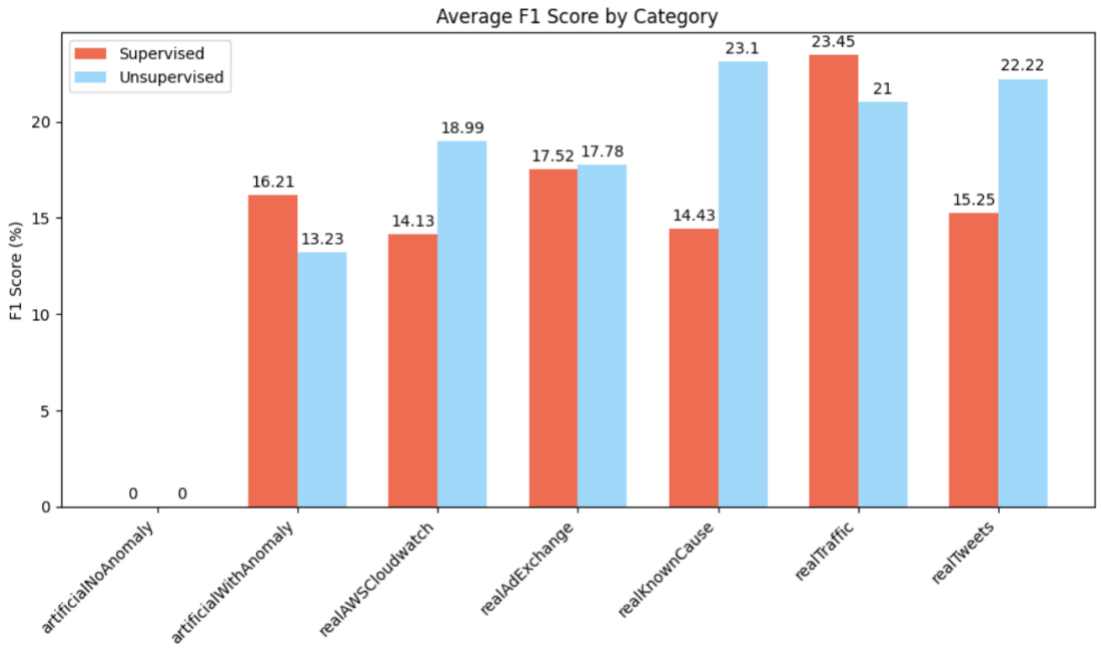

Figure 2

Figure 3 shows average precision for supervised model does not differ a lot from unsupervised in some cases and even having higher for artificial data.

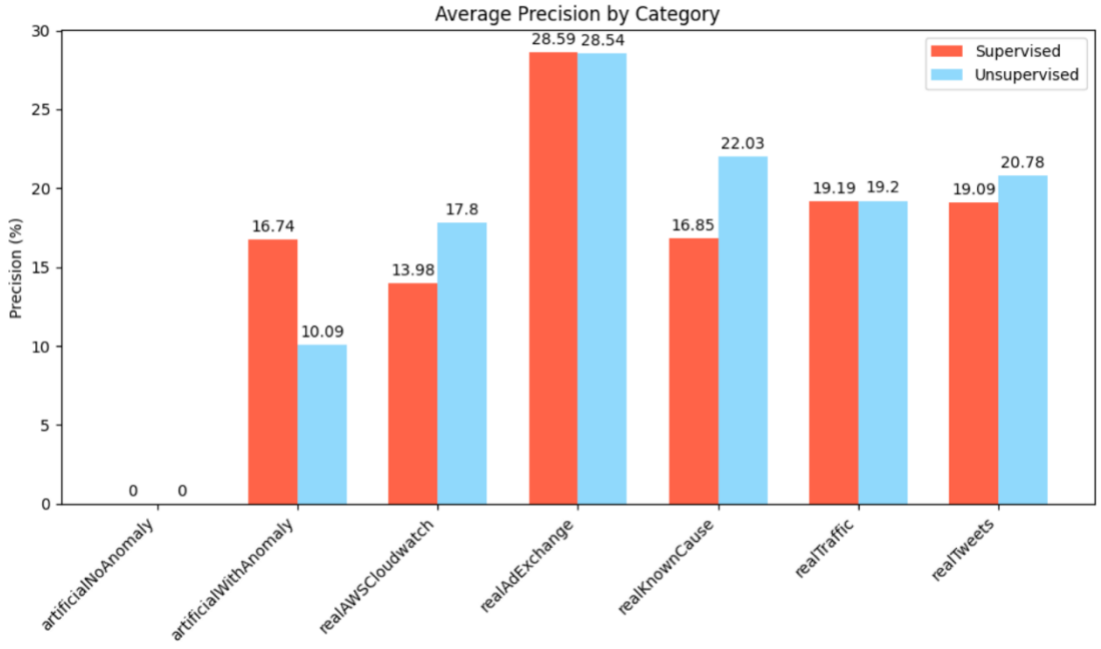

Figure 3

Figure 4 shows average recall for unsupervised model has a clear difference in recalls meaning it was more successful at catching anomalies.

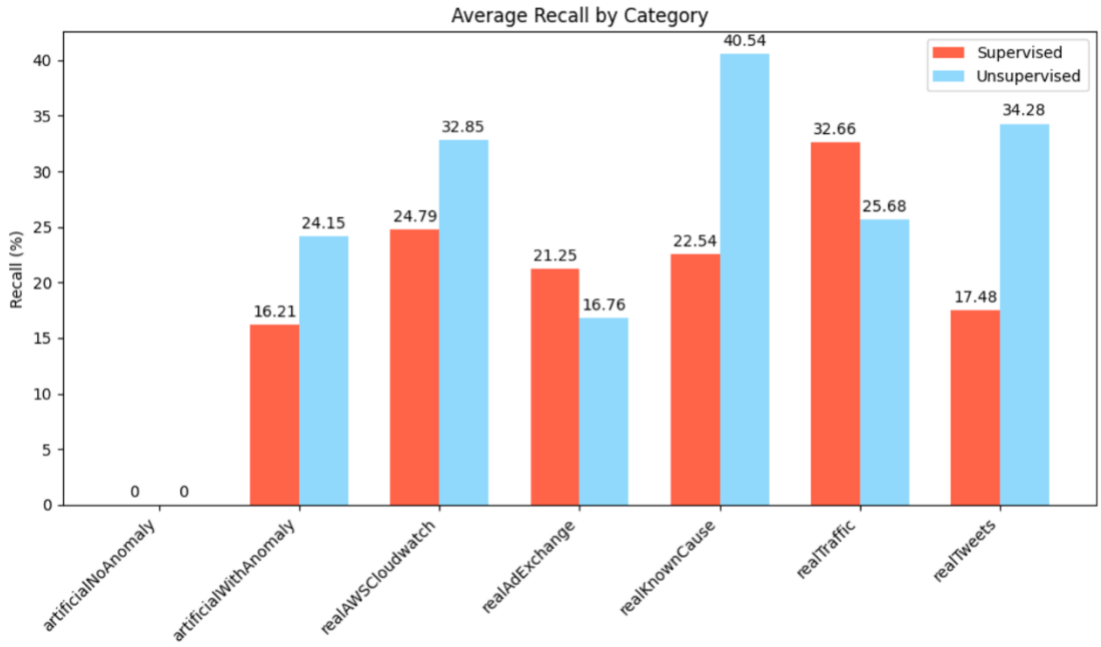

Figure 4

Figure 5 shows the overall average of all 58 files. The recall and F1 score is clearly better for unsupervised with only slight difference in precision with supervised model.

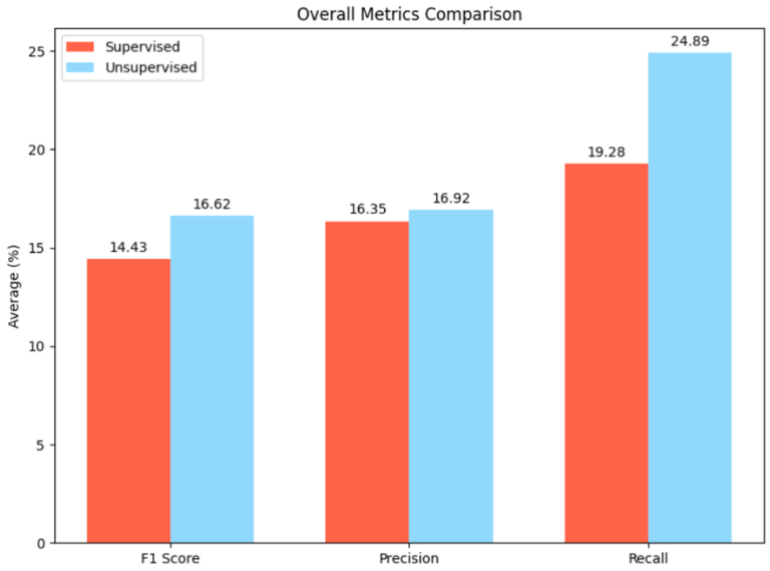

Figure 5

5. Conclusion

This project explored the use of LSTM models with conformal prediction to detect anomalies in time series data from the Numenta Anomaly Benchmark. Two different approaches were implemented in Julia with custom configuration/parameters for all 58 files. The unsupervised model used next-step forecasting and then check if it is expected or not, and the supervised model would directly classify using labelled anomaly sequences. Both models used conformal prediction to calculate thresholds and avoid manual guessing.

The results showed that both models provide a good starting point for more experimentation. The unsupervised model showed higher recall and F1 scores across most NAB categories. It was more effective at detecting real anomalies in noisy real world data. It also had a higher false positive rate, reducing its precision. The supervised model achieved better precision, especially in aritifical datasets, but also failed to detect a lot of the anomalies, resulting in lower recall.

Overall, the unsupervised model proved to be more robust and general across different types of data. Its ability to learn without the need of labels and not knowing what an anomaly actually looks like makes it a decent option for real-time anomaly detection scenarios where labels are very rare or unavailable. The supervised model is still where high precision is our goal and labels are detailed and properly available

Limitations and Future Work

While this project successfully implemented and compared supervised and unsupervised LSTM models with conformal prediction, there are still many improvements that can be made.

One clear limitation is the very basic and simple architecture with fixed hyperparameters. More layers could be added or implementation of bidirectional LSTM's could help improve results by a lot. Using transformers could also help recognising more complex patterns.

Hyperparameter tuning was done manually, which did help improve results by a lot but was very time consuming and not scalable. Future work could automate this process using techniques like grid search, Bayesian optimization, or even meta-learning.

Further improvements can be made by trying advanced conformal techniques, like online updating of thresholds. The unsupervised model in had strong recall but also gave lots of false positives. Future improvements could include methods to group anomaly spikes, apply secondary filters, or use contextual cues to reduce unnecessary alerts and make the model more practical for real-world deployments.

Finally, both models currently train on each dataset separately. We could try transfer learning or shared pretraining across multiple datasets to help create more general models.

6. Code

The full anomaly detection pipeline was implemented in Julia, using core packages such as `Flux.jl` for deep learning, `JSON.jl` for configuration loading, and `CSV.jl/DataFrames.jl` for handling input files.
The code is organized into clear sections to ensure reusability across all 58 datasets in the NAB benchmark.

6.1 Dataset & Colab Notebook

Dataset: NAB GitHub
The complete code is also available on Google Colab to provide more structured/organized view.
Google Colab: Notebook

6.2 Parameter Loading

A configuration JSON file (params_{model-type}.json) was used to store dataset-specific parameters.

```
function load_params(json_path::String)::Dict{String, Dict{String, Float64}}
    if isfile(json_path)
        return JSON.parsefile(json_path)
    else
        error("params_supervised.json not found at $json_path")
    end
end
```

Sample from JSON files:

```
},
"realTraffic/occupancy_t4013.csv": {
    "probation_ratio": 0.15,
    "calib_ratio": 0.1,
    "window_size": 10,
    "epochs": 40,
    "lr": 0.0008,
    "conf": 0.90,
    "dropout": 0.3
},
"realAdExchange/exchange-2_cpm_results.csv": {
    "probation_ratio": 0.15,
    "calib_ratio": 0.1,
    "window_size": 10,
    "epochs": 25,
    "lr": 0.001,
    "conf": 0.85,
    "dropout": 0.25
},
```

```
},
"realTraffic/occupancy_t4013.csv": {
  "train_ratio": 0.1,
  "calib_ratio": 0.05,
  "window_size": 30,
  "epochs": 30,
  "lr": 0.0007,
  "dropout": 0.1,
  "conf": 0.85
},
"realTraffic/speed_6005.csv": {
  "train_ratio": 0.1,
  "calib_ratio": 0.05,
  "window_size": 10,
  "epochs": 40,
  "lr": 0.0008,
  "dropout": 0.3,
  "conf": 0.90
},
```

6.3 Data Loading and Preprocessing

For unsupervised model only the timestamps, values are loaded, but for supervised labels are also loaded.

```
function load_data_with_labels(filepath::String, labelpath::String)
    all_labels = JSON.parsefile(labelpath)
    dataset_key = joinpath(splitpath(filepath)[end-1:end]...)

    df = CSV.read(filepath, DataFrame)
    timestamps = DateTime.(df.timestamp, dateformat"yyyy-MM-dd HH:MM:SS")
    values = Float32.(df.value)

    anomaly_windows = get(all_labels, dataset_key, [])

    labels = Vector{Int}(undef, length(timestamps))
    for (i, ts) in enumerate(timestamps)
        labels[i] = any(DateTime(w[1], dateformat"yyyy-MM-dd HH:MM:SS.ssss") <= ts <=
        DateTime(w[2], dateformat"yyyy-MM-dd HH:MM:SS.ssss") for w in anomaly_windows) ? 1 : 0
    end

    return timestamps, values, labels
end
```

Supervised splitting and normalizing:

Splits the dataset into train, calibration, and test sets using preset ratios, and normalizes values using the training portion only

```
function split_and_normalize(values, labels,
    train_ratio::Float64, calib_ratio::Float64)

    n = length(values)
    train_end = floor(Int, n * train_ratio)
    calib_end = floor(Int, n * (train_ratio + calib_ratio))

    train_raw = values[1:train_end]
    mean_train = mean(train_raw)
    std_train = std(train_raw)
    normal_values = (values .- mean_train) ./ std_train

    train_values = normal_values[1:train_end]
    train_labels = labels[1:train_end]

    calib_values = normal_values[train_end+1:calib_end]
    calib_labels = labels[train_end+1:calib_end]

    test_values = normal_values[calib_end+1:end]
    test_labels = labels[calib_end+1:end]

    return (;train_values, train_labels,
            calib_values, calib_labels,
            test_values, test_labels)
end
```

Unsupervised splitting and normalizing:

Extracts a probation window from early data, splits it into train and calibration sets, and uses only the training part to normalize the full sequence for evaluation.

```
function split_and_normalize(values::Vector{Float32}, prob_ratio::Float64, calib_ratio::Float64)
    n = length(values)
    prob_count = Int(floor(prob_ratio * n))
    calib_count = Int(floor(calib_ratio * prob_count))
    train_count = prob_count - calib_count

    probation_vals = values[1:prob_count]
    mean_prob = mean(probation_vals)
    std_prob = std(probation_vals)
    normal_values = (values .- mean_prob) ./ std_prob

    train_data = normal_values[1:train_count]
    calib_data = normal_values[train_count+1 : prob_count]
    test_data  = normal_values[prob_count+1 : end]

    return (;train_data, calib_data, test_data)
end
```

6.4 Data Sequencing

Splits the data into short overlapping sequences so the model can learn patterns. This is same for both models.

```
function split_windows(values::Vector{Float32}, window_size::Int)
    n_seq = length(values) - window_size
    x = Array{Float32}(undef, window_size, 1, n_seq)
    y = Array{Float32}(undef, 1, n_seq)

    for i in 1:n_seq
        x[:, 1, i] = values[i : i + window_size - 1]
        y[1, i] = values[i + window_size]
    end

    return x, y
end
```

6.5 Model Training

Supervised Model Training:

Trains the model to classify if a sequence ends in an anomaly using binary cross-entropy loss and sigmoid activation.

```
function setup_model(window_size::Int, dropout::Float64)
    return Chain(
        LSTM(window_size => 64),
        Dropout(dropout),
        Dense(64 => 1),
        sigmoid
    )
end

function train_model(x_train::Array{Float32, 3}, y_train::Array{Float32,2};
    window_size::Int, epochs::Int, lr::Float64, dropout::Float64)

    model = setup_model(window_size, dropout)
    opt_state = Flux.setup(ADAMW(lr), model)

    for epoch in 1:epochs
        Flux.reset!(model)

        loss, grads = Flux.withgradient(model) do m
            y_pred = m(x_train)
            y_true = reshape(y_train, size(y_pred))
            Flux.logitbinarycrossentropy(y_pred, y_true)
        end

        Flux.update!(opt_state, model, grads[1])
        println("Epoch $epoch -- Loss: $(round(loss, digits=4))")
    end

    return model
end
```

Unsupervised Model Training:

Trains the model to predict the next value in a sequence using Mean Squared Error, helping it learn normal behaviour for anomaly detection.

```
function setup_model(window_size::Int, dropout::Float64)
    return Chain(
        LSTM(window_size => 64),
        Dropout(dropout),
        Dense(64 => 1)
    )
end

function train_model(x_train::Array{Float32, 3}, y_train::Array{Float32,2};
    window_size::Int, epochs::Int, lr::Float64, dropout::Float64)

    model = setup_model(window_size, dropout)
    opt_state = Flux.setup(ADAMW(lr), model)

    for epoch in 1:epochs
        Flux.reset!(model)

        loss, grads = Flux.withgradient(model) do m
            y_pred = m(x_train)
            y_pred = dropdims(y_pred, dims=2)
            Flux.Losses.mse(y_pred, y_train)
        end

        Flux.update!(opt_state, model, grads[1])
        println("Epoch $epoch -- Loss = $(round(loss, digits=4))")
    end

    return model
end
```

6.6 Conformal Prediction and Evaluation

Supervised:

Calculates residuals based on the model's predicted probability vs. true labels in the calibration set. A threshold is set at a chosen confidence level, and any test prediction with error above this threshold is flagged as an anomaly.

```
function conformal_threshold(model, x_calib, y_calib, x_test, confidence::Float64)
    Flux.reset!(model)
    calib_prob = model(x_calib)
    residuals = vec(abs.(calib_prob .- y_calib))

    if any(isnan, residuals)
        return NaN, NaN, NaN
    end

    threshold = quantile(residuals, confidence)

    println("\nThreshold = $(round(threshold, digits=4))")

    Flux.reset!(model)
    test_prob = model(x_test)
    test_residuals = abs.(test_prob .- 0.0)
    predicted_anomalies = test_residuals .> threshold

    return predicted_anomalies, test_prob, threshold

end
function calc_metrics(filename:: String, y_true::Vector{Int}, y_pred::BitVector)
    TP = sum((y_true .== 1) .& (y_pred .== true))
    FP = sum((y_true .== 0) .& (y_pred .== true))
    FN = sum((y_true .== 1) .& (y_pred .== false))

    precision = TP + FP == 0 ? 0.0 : TP / (TP + FP)
    recall = TP + FN == 0 ? 0.0 : TP / (TP + FN)
    f1 = precision + recall == 0 ? 0.0 : 2 * (precision * recall) / (precision + recall)

    println("\nEvaluation Metrics for $filename")
    println("TP = $TP | FP = $FP | FN = $FN")
    println("Precision: $(round(precision * 100, digits=2))%")
    println("Recall: $(round(recall * 100, digits=2))%")
    println("F1 Score: $(round(f1 * 100, digits=2))%")

    return (;precision, recall, f1, TP, FP, FN)
end
```

Unsupervised:

Uses residuals from the calibration set to set a threshold. During testing, if the model's prediction error exceeds this threshold, the point is marked as an anomaly.

```julia
function conformal_classification(model, x_calib, y_calib; confidence::Float64)
    residuals = Float32[]

    for i in 1:size(x_calib, 3)
        x = x_calib[:, :, i]
        Flux.reset!(model)
        y_pred = model(x)
        push!(residuals, abs(y_pred[1] - y_calib[1, i]))
    end

    if any(isnan, residuals)
        return NaN, residuals
    end

    threshold = quantile(residuals, confidence)

    println("Threshold = ", round(threshold, digits=4))
    return threshold, residuals
end
```

```julia
function evaluate_model(model, x_eval, y_eval, threshold)
    residuals = Float32[]
    predictions = Float32[]
    anomaly_flags = Int[]

    for i in 1:size(x_eval, 3)
        x = x_eval[:, :, i]
        Flux.reset!(model)
        y_pred = model(x)[1]
        res = abs(y_pred - y_eval[1, i])

        push!(residuals, res)
        push!(predictions, y_pred)
        push!(anomaly_flags, res > threshold ? 1 : 0)
    end

    return (;predictions, residuals, anomaly_flags)
end

function load_true_labels(label_path::String, file_key::String,
    eval_timestamps::Vector{DateTime})

    labels = JSON.parsefile(label_path)
    windows = labels[file_key]

    anomaly_windows = [(DateTime(w[1], dateformat"yyyy-MM-dd HH:MM:SS.ssss"),
    DateTime(w[2], dateformat"yyyy-MM-dd HH:MM:SS.ssss")) for w in windows]

    true_flags = Int[]
    for ts in eval_timestamps
        is_anomaly = any(r -> ts ≥ r[1] && ts ≤ r[2], anomaly_windows)
        push!(true_flags, is_anomaly ? 1 : 0)
    end

    return true_flags
end

function calc_metrics(filename::String, pred_flags::Vector{Int}, true_flags::Vector{Int})
    TP = sum((pred_flags .== 1) .& (true_flags .== 1))
    FP = sum((pred_flags .== 1) .& (true_flags .== 0))
    FN = sum((pred_flags .== 0) .& (true_flags .== 1))

    precision = TP + FP == 0 ? 0.0 : TP / (TP + FP)
    recall    = TP + FN == 0 ? 0.0 : TP / (TP + FN)
    f1        = precision + recall == 0 ? 0.0 : 2 * (precision * recall) / (precision + recall)

    println("\nEvaluation Metrics for $filename")
    println("TP = $TP | FP = $FP | FN = $FN")
    println("Precision: $(round(precision * 100, digits=2))%")
    println("Recall: $(round(recall * 100, digits=2))%")
    println("F1 Score: $(round(f1 * 100, digits=2))%")

    return (;precision, recall, f1, TP, FP, FN)
end
```

6.7 Implementation

Running the supervised model pipeline

```
data_root = "data"
label_path = "labels/combined_windows.json"
param_path = "params_supervised.json"
      You, last week • Initial CA2 stuff
params = load_params(param_path)

all_files = sort(collect(keys(params)))

for filename in all_files
    println("\nProcessing: $filename")

    file_path = joinpath(data_root, filename)

    # Load data and parameters
    timestamps, values, labels = load_data_with_labels(file_path, label_path)
    config = params[filename]
    train_ratio = config["train_ratio"]
    calib_ratio = config["calib_ratio"]
    window_size = Int(config["window_size"])
    epochs = Int(config["epochs"])
    lr = Float64(config["lr"])
    dropout = Float64(config["dropout"])
    conf = Float64(config["conf"])

    # Split and normalize data
    clean_data = split_and_normalize(values, labels, train_ratio, calib_ratio)

    # Create sequences
    x_train, y_train = split_windows(clean_data.train_values, clean_data.train_labels, window_size)
    x_calib, y_calib = split_windows(clean_data.calib_values, clean_data.calib_labels, window_size)
    x_test, y_test = split_windows(clean_data.test_values, clean_data.test_labels, window_size)

    # Train model
    model = train_model(
        x_train, y_train;
        window_size=window_size,
        epochs=epochs,
        lr=lr,
        dropout=dropout
    )

    # Classification
    y_pred_class, test_prob, threshold = conformal_threshold(
        model, x_calib, y_calib, x_test, conf)

    if !isfinite(threshold)
        save_metrics(filename, (;TP=0, FP=0, FN=0, precision=0.0, recall=0.0, f1=0.0, mse=0.0, mae=0.0))
        continue
    end

    # Evaluation
    y_true = Int.(vec(y_test))
    y_pred = vec(y_pred_class)
    metrics = calc_metrics(filename, y_true, y_pred)

    # Save metrics
    save_metrics(filename, metrics)
end
println("\nAll files processed successfully")
```

Running the unsupervised model pipeline

```
data_root = "data"
label_path = "labels/combined_windows.json"
param_path = "params_unsupervised.json"

params = load_params(param_path)

all_files = sort(collect(keys(params)))

for filename in all_files
    println("\nProcessing: $filename")

    file_path = joinpath(data_root, filename)

    # Load data and parameters
    ts, values = load_csv_data(file_path)
    config = params[filename]
    probation_ratio = config["probation_ratio"]
    calib_ratio = config["calib_ratio"]
    window_size = Int(config["window_size"])
    epochs = Int(config["epochs"])
    lr = Float64(config["lr"])
    dropout = Float64(config["dropout"])
    conf = Float64(config["conf"])

    # Split and normalize data
    train_data, calib_data, eval_data = split_and_normalize(values, probation_ratio, calib_ratio)

    # Create sequences
    x_train, y_train = split_windows(train_data, window_size)
    x_calib, y_calib = split_windows(calib_data, window_size)
    x_eval,  y_eval  = split_windows(eval_data,  window_size)

    # Train model
    model = train_model(
        x_train, y_train;
        window_size=window_size,
        epochs=epochs,
        lr=lr,
        dropout=dropout
    )

    # Classification
    threshold, calib_residuals = conformal_classification(
        model, x_calib, y_calib; confidence=conf)

    if !isfinite(threshold)
        save_metrics(filename, (;TP=0, FP=0, FN=0, precision=0.0, recall=0.0, f1=0.0, mse=0.0, mae=0.0))
        continue
    end

    # Evaluate
    eval_results = evaluate_model(model, x_eval, y_eval, threshold)

    # True labels & metrics
    n = length(values)
    n_prob = Int(floor(probation_ratio * n))
    ts_eval = ts[n_prob + window_size + 1 : end]

    true_flags = load_true_labels(label_path, filename, ts_eval)
    metrics = calc_metrics(filename, eval_results.anomaly_flags, true_flags)

    # Save model & metrics
    # save_model(model, filename)
    save_metrics(filename, metrics)
end

println("\nAll files processed successfully")
```

7. References

Ahmad, S., Lavin, A., Purdy, S., & Agha, Z. (2017). Unsupervised real-time anomaly detection for streaming data. *Neurocomputing, 262*, 134–147. https://doi.org/10.1016/j.neucom.2017.04.070

Amazon Web Services. (n.d.). *Amazon Lookout for Metrics*. https://aws.amazon.com/lookout-for-metrics/

Cui, Y., Ahmad, S., & Hawkins, J. (2016). Continuous online sequence learning with an unsupervised neural network model. *Neural Computation, 28*(11), 2474–2504. https://doi.org/10.1162/NECO_a_00893

Google Cloud. (n.d.). *Vertex AI*. https://cloud.google.com/vertex-ai

Hundman, K., Constantinou, V., Laporte, C., Colwell, I., & Soderstrom, T. (2018). Detecting spacecraft anomalies using LSTMs and nonparametric dynamic thresholding. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (pp. 387–395). https://doi.org/10.1145/3219819.3219845

Microsoft. (n.d.). *Azure Anomaly Detector*. https://azure.microsoft.com/en-us/services/cognitive-services/anomaly-detector/