Digital Twin of 5G Network

Technical documentation

Dávid Truhlář

GETTING STARTED

CHAPTER

ONE

GETTING STARTED

Welcome to the documentation for the **Digital Twin of a 5G Network** developed as part of a bachelor thesis at the Slovak University of Technology. This section will guide you through setting up, running, and extending the digital twin environment.

1.1 Overview

This project simulates and analyzes 5G network behavior by integrating the following components:

- **Open5GS** – a full 5G core implementation.
- **UERANSIM** – emulation of multiple UE and gNBs.
- **Prometheus & Grafana** – monitoring and visualization.
- **LSTM models** – classification of network behavior using deep learning.
- **Sphinx documentation** – modular API docs.

1.2 Prerequisites

Before running the project, ensure you have:

- Docker & Docker Compose
- Python 3.9+ and *venv*
- Git
- At least 16 GB RAM (recommended for full stack simulation)

CHAPTER TWO

INSTALLATION GUIDE

First, ensure you have the following prerequisites installed:

```
Docker and Docker Compose
Python 3.9+ and `venv`
Git
```

Second, clone the repository:

```
git clone https://github.com/xtruhlar/5GDigitalTwin.git
cd 5GDigitalTwin/Implementation
```

To build docker images

```
cd ./open5gs/base
docker build -t docker_open5gs .

cd ../ueransim
docker build -t docker_ueransim .

cd ..
```

To set the environment variables

```
cp .env.example .env

set -a
source .env
set +a
```

To run the project, navigate to *Implementation/* and execute the following command

```
docker compose -f deploy-all.yaml up --build -d
```

To add subscribers to Open5GS core, run the following commands

```
docker exec -it mongo mkdir -p /data/backup
ocker cp ./open5gs/mongodb_backup/open5gs mongo:/data/backup/open5gs
docker exec -it mongo mongorestore --uri="mongodb://localhost:27017" --db open5gs /data/
↪backup/open5gs
```

To ensure everything works properly, open http://localhost:9999/ in your browser and login using credentials:

```
Username: admin
Password: 1423
```

To connect UERANSIM gNB to Open5GS, run the following command

```
docker compose -f nr-gnb.yaml -p gnodeb up -d && docker container attach nr_gnb
```

To connect UERANSIM UE to Open5GS, run the following command

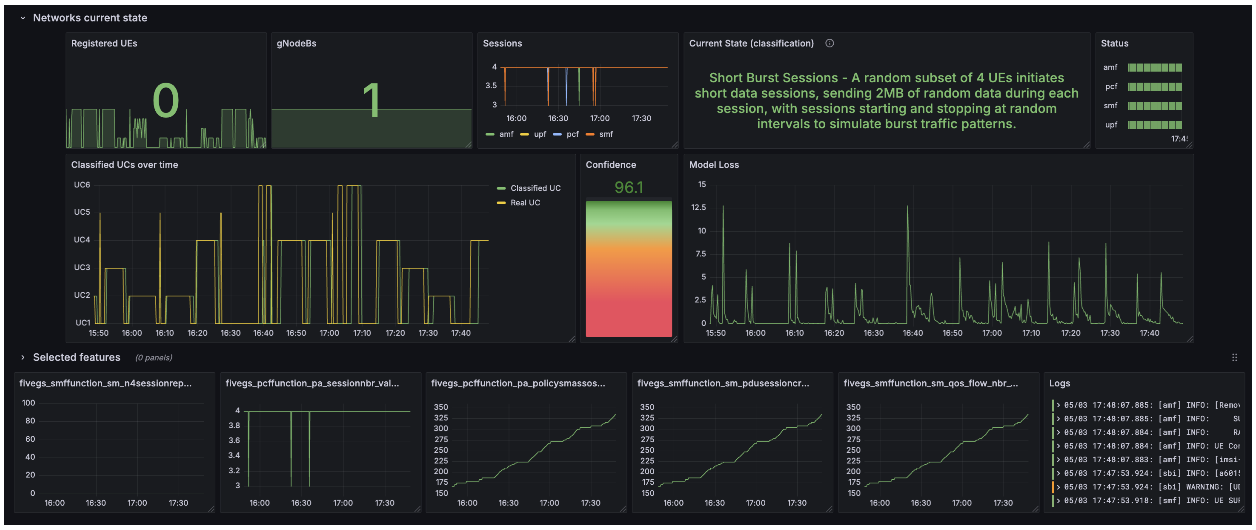
```
docker compose -f nr-ue.yaml -p ue up -d && docker container attach ue
```

Then go to Grafana, open http://localhost:3000/ in your browser and login using credentials:

```
Username: open5gs
Password: open5gs
```

Open menu on the left, click on *Dashboards*. Select *Current state Dash* and you can see the current state of your 5G network.

Example:


Networks current state
Registered UEs
0
gNodeBs
1
Sessions
amf
upf
pcf
smf
Current State (classification)
Status
amf
pcf
smf
upf
Classified UCs over time
Classified UC
Real UC
Confidence
96.1
Model Loss
Selected features
(0 panels)
fivegs_smffunction_sm_n4sessionrep...
fivegs_pcffunction_pa_sessionnbr_val...
fivegs_pcffunction_pa_policysmassos...
fivegs_smffunction_sm_pdusessioncr...
fivegs_smffunction_sm_qos_flow_nbr_...
Logs

Short Burst Sessions - A random subset of 4 UEs initiates short data sessions, sending 2MB of random data during each session, with sessions starting and stopping at random intervals to simulate burst traffic patterns.

```
> 05/03 17:48:07.885: [amf] INFO: [Remo
> 05/03 17:48:07.885: [amf] INFO:     SU
> 05/03 17:48:07.884: [amf] INFO:     RA
> 05/03 17:48:07.884: [amf] INFO: UE Co
> 05/03 17:48:07.883: [amf] INFO: [imsi-
> 05/03 17:47:53.924: [sbi] INFO: [a601
> 05/03 17:47:53.924: [sbi] WARNING: [U
> 05/03 17:47:53.918: [smf] INFO: UE SU
```

Fig. 1: Grafana dashboard

CHAPTER THREE

NORMAL SURFING

`uc1.run_uc1()`

Run UC1: Normal Surfing scenario.

Simulates user behavior with intermittent UE connectivity and randomized data downloads to mimic typical mobile web browsing patterns.

Scenario Summary

- Starts half of the UEs.
- **UEs may**
  - download random chunks of data (5–50MB),
  - randomly disconnect or reconnect during the session.
- The scenario runs for a randomized session duration (60–600 seconds).
- Active UEs are defined in *nr-ue{i}.yaml* Docker Compose files.
- Writes the current UC label to *data/current_uc.txt*.

Args

None

Returns

None

CHAPTER

FOUR

VIDEO STREAMING

`uc2.run_uc2()`

Run UC2: Video Streaming scenario.

Simulates a typical video streaming session where all UEs continuously receive data. This scenario helps to evaluate throughput and session stability under constant load.

Scenario Summary

- Starts 4 UEs using Docker Compose.
- Each UE downloads 2MB of random data every second.
- Streaming duration is randomized between 300 and 600 seconds.
- The UC label is logged into 'data/current_uc.txt'.

Args

None

Returns

None

CHAPTER

FIVE

PERIODIC KEEP-ALIVE

uc3.**run_uc3**()

Run UC3: Periodic Keep-Alive scenario.

Simulates multiple UEs that periodically send HTTP requests (keep-alive pings) to a remote server. This pattern reflects real-world background traffic in mobile applications (e.g. chat apps, weather updates).

Scenario Summary

- Starts 4 UEs as containers using Docker Compose.
- Each UE sends periodic HTTP GET requests (via *curl*) to a predefined URL.
- The interval between pings is randomized between 30–35 seconds.
- Simulation duration is randomly set between 300–600 seconds.
- Scenario type is logged into 'data/current_uc.txt'.

Args

None

Returns

None

CHAPTER

SIX

SHORT BURST SESSIONS

`uc4.run_uc4()`

Run UC4: Short Burst Sessions scenario.

Simulates multiple UEs initiating brief data sessions at random intervals. This pattern mimics sporadic, high-frequency user actions (e.g. short API requests, fast-loading web content) in mobile networks.

Scenario Summary

- Randomly selects a UE out of 4 available.
- Starts a short Docker container session for the selected UE.
- Simulates a 2MB data transfer using *dd* inside the container.
- After 2–4 seconds, the UE container is stopped.
- Waits 3–6 seconds and repeats until the scenario duration ends.
- Marks current use case in ‘data/current_uc.txt’.

Args
None

Returns
None

CHAPTER

SEVEN

LOAD REGISTRATION ANOMALY

`uc5.run_uc5()`

Run UC5: Load Registration Anomaly scenario.

Simulates a stress event in the 5G network by concurrently connecting multiple UEs. This tests the network's ability to handle sudden, simultaneous registration attempts — a common anomaly in overloaded environments.

Scenario Summary

- Starts 4 UE containers simultaneously using *subprocess.Popen*.
- Waits for all containers to fully initialize.
- Holds all UE sessions active for a short time (5 seconds by default).
- Stops all containers simultaneously after the wait period.
- Logs scenario activity to 'data/current_uc.txt'.

Args

None

Returns

None

CHAPTER EIGHT

AUTHENTICATION FAILURE ALERT

`uc6.run_uc6()`

Run UC6: Authentication Failure Alert scenario.

Simulates an authentication failure event in a 5G network by repeatedly starting a misconfigured UE (User Equipment) that fails to register due to incorrect credentials or malformed configuration. This scenario is useful for testing network response to repeated failed attempts and monitoring for anomaly detection mechanisms.

Scenario Summary

- Launches a specially prepared UE (ID=100) which is expected to fail authentication.
- Repeats the process a random number of times (3 to 6 retries).
- Between each attempt, the UE is stopped and a random interval (5–30 seconds) is observed.
- Total scenario duration is also bounded by a global timeout (120–300 seconds).
- All events are logged to the console and scenario type is saved to *data/current_uc.txt*.

Args

None

Returns

None

CHAPTER NINE

LSTM MODEL WITH ATTENTION

class `lstm_attention_model.`**`AttentionLayer`**(**args*, ***kwargs*)

Bases: `Layer`

Custom attention layer compatible with LSTM outputs. Outputs a weighted sum across the time dimension.

Source: https://www.geeksforgeeks.org/adding-attention-layer-to-a-bi-lstm/?

`build`(*input_shape*)

Build the attention layer.

Args

- input_shape: Shape of the input tensor.

Returns

None

`call`(*x*)

Apply the attention mechanism to the input tensor.

Args

- x: Input tensor of shape (batch_size, timesteps, features).

Returns

- output: Weighted sum of the input tensor across the time dimension.

`compute_output_shape`(*input_shape*)

Compute the output shape of the attention layer.

Args

- input_shape: Shape of the input tensor.

Returns

- output_shape: Shape of the output tensor.

`lstm_attention_model.`**`build_attention_model`**`(`*input_shape*, *num_classes*`)`

Build and return an attention-based LSTM model.

Args

- input_shape: tuple, shape of the input data (timesteps, features)
- num_classes: int, number of output classes

Returns

- Keras Model instance

`lstm_attention_model.`**`train_attention_model`**`()`

LSTM model with custom attention mechanism for multi-class classification of time-series data.

This module defines a deep learning model using TensorFlow and Keras, integrates a custom attention mechanism, and trains the model on preprocessed input data with categorical labels.

Expected data format:

- Input: X_train.npy, X_test.npy (shape: [samples, 60, features])
- Labels: y_train.npy, y_test.npy (categorical class indices)
- Class weights: class_weights.json

The trained model is saved as HDF5 and Keras formats.

CHAPTER

TEN

LSTM BATHNORM MODEL

`lstm_bathnorm_model.train_batchnorm_model()`

LSTM BatchNorm Model

This module defines a deep learning model using TensorFlow and Keras with Batch Normalization for multi-class classification of time-series data.

Expected data format:

- Input: X_train.npy, X_test.npy (shape: [samples, 60, features])
- Labels: y_train.npy, y_test.npy (categorical class indices)
- Class weights: class_weights.json

The trained model is saved as HDF5 and Keras formats.

CHAPTER

ELEVEN

LSTM BASE MODEL

`lstm_base_model.`**`build_base_model`**`(`*X_train*, *y_train*, *X_test*, *y_test*, *class_weight_dict*`)`

LSTM Base Model for classification of network use case scenarios.

This script loads preprocessed training and testing data, defines and trains a baseline LSTM model, evaluates its performance, and saves the final model in HDF5 and Keras formats. Class balancing is handled using precomputed class weights.

Usage

- This script is designed to be executed as a module. Use the function *build_base_model()* to construct and optionally train the model.

Args

- X_train (numpy.ndarray): Preprocessed training data.
- y_train (numpy.ndarray): Labels for the training data.
- X_test (numpy.ndarray): Preprocessed testing data.
- y_test (numpy.ndarray): Labels for the testing data.
- class_weight_dict (dict): Class weights for handling class imbalance.

Returns

- model (tensorflow.keras.Model): Trained LSTM model.

CHAPTER

TWELVE

LSTM ROBUST MODEL

`lstm_robust_model.`**`build_robust_model`**`()`

Trains a robust LSTM model and saves it to the *trained_models/* directory. The model consists of 3 LSTM layers with varying dropout rates and 2 Dense layers. It is designed for classifying UC classes based on preprocessed sequential inputs.

Args

None

Returns

None

The function saves the trained model to the *trained_models/* directory and generates visualizations for the confusion matrix and training history.

CHAPTER

THIRTEEN

CLASSIFY THE REAL DATA USING LSTM

Module for evaluating multiple LSTM models (base, robust, batchnorm, attention) on real-world 5G network data, with the option to fine-tune the attention model.

This script includes

- Definition of a custom attention layer,
- Real-data class weight computation,
- Sequence generation for LSTM input,
- Model accuracy evaluation via classification report,
- Optional fine-tuning of the attention-based LSTM model.

`class lstm_results_real_data.`**`AttentionLayer`**`(`*`*args`*`, `*`**kwargs`*`)`

Bases: `Layer`

Custom attention layer compatible with LSTM outputs. Outputs a weighted sum across the time dimension.

Source: https://www.geeksforgeeks.org/adding-attention-layer-to-a-bi-lstm/?

`build``(`*`input_shape`*`)`

Build the attention layer.

Args

- input_shape: Shape of the input tensor.

Returns

None

`call``(`*`x`*`)`

Apply the attention mechanism to the input tensor.

Args

- x: Input tensor of shape (batch_size, timesteps, features).

Returns

- output: Weighted sum of the input tensor across the time dimension.

`compute_output_shape``(`*`input_shape`*`)`

Compute the output shape of the attention layer.

Args

- input_shape: Shape of the input tensor.

Returns

- output_shape: Shape of the output tensor.

`lstm_results_real_data.`**`create_sequences`**`(`*X*, *y*, *seq_len=60*`)`

Converts flattened input arrays into sliding window sequences for LSTM input.

Args

- X (np.ndarray): Input features of shape (samples, features).
- y (np.ndarray): Target labels corresponding to input samples.
- seq_len (int): Length of each sequence window. Default is 60.

Returns

- tuple: (X_seq, y_seq) where X_seq has shape (samples - seq_len, seq_len, features) and y_seq has shape (samples - seq_len,).

`lstm_results_real_data.`**`evaluate_model`**`(`*model*, *X_seq*, *y_seq*, *name*`)`

Evaluates a trained model on given sequential data and prints classification report.

Args

- model (keras.Model): The trained Keras model to evaluate.
- X_seq (np.ndarray): Sequential input data (samples, seq_len, features).
- y_seq (np.ndarray): True class labels.
- name (str): Name of the model for display purposes.

Returns

None

`lstm_results_real_data.`**`load_and_preprocess_data`**`()`

Loads the real-world labeled dataset and applies categorical mappings.

Args

None

Returns

- pd.DataFrame: Preprocessed dataset with mapped categorical columns and timestamps.

`lstm_results_real_data.`**`run_evaluation_and_finetuning`**`()`

Main function to evaluate four trained LSTM models and fine-tune the attention model using real labeled data.

Args

None

Returns

None

CHAPTER

FOURTEEN

EXPLORATORY DATA ANALYSIS

EDA module for exploratory analysis of synthetic and real 5G network datasets.

This module contains functions to load data, preprocess it, visualize it, and perform feature selection using multiple strategies including RF, RFE, RFECV, SFS and permutation importance.

Functions in this module should be called explicitly from a main script or notebook.

`eda.compute_class_weights(`*y*`)`

Compute class weights for imbalanced classes.

Args

- y (pd.Series): Target variable.

Returns

- dict: Dictionary mapping class labels to their corresponding weights.

`eda.load_dataset(`*path*`)`

Load dataset from CSV file.

Return type
`DataFrame`

Args

- path (str): Path to the CSV file.

Returns

- pd.DataFrame: Loaded dataset.

`eda.load_maps(`*log_map_path='./json/log_map.json', app_map_path='./json/app_map.json', uc_map_path='./json/uc_map.json'*`)`

Load mapping dictionaries from JSON files.

Args

- log_map_path (str): Path to the log type mapping JSON file.
- app_map_path (str): Path to the application mapping JSON file.
- uc_map_path (str): Path to the use case mapping JSON file.

Returns

- **tuple: A tuple containing three dictionaries:**
  - log_map (dict): Mapping of log types to integers.

    - app_map (dict): Mapping of applications to integers.
    - uc_map (dict): Mapping of use cases to integers.

`eda.`**`permutation_importance_stable`**`(`*X*, *y*, *selected_features*, *n_runs=10*`)`

Calculate stable permutation importances over multiple runs.

Args

- X (pd.DataFrame): Feature matrix.
- y (pd.Series): Target variable.
- selected_features (list): List of selected feature names.
- n_runs (int): Number of runs for stability.

Returns

- pd.DataFrame: DataFrame containing mean and std of importances.

`eda.`**`preprocess_data`**`(`*df*, *log_map*, *app_map*, *uc_map*`)`

Preprocess dataset: fill NA, map strings to ints, scale numeric columns.

Args

- df (pd.DataFrame): Input DataFrame to preprocess.
- log_map (dict): Mapping of log types to integers.
- app_map (dict): Mapping of applications to integers.
- uc_map (dict): Mapping of use cases to integers.

Returns

- **tuple: A tuple containing:**
    - X_scaled (np.ndarray): Scaled feature matrix.
    - X (pd.DataFrame): Original feature matrix.
    - y (pd.Series): Target variable.

`eda.`**`random_forest_importance`**`(`*X_scaled*, *X*, *y*`)`

Train Random Forest and return feature importances.

Args

- X_scaled (np.ndarray): Scaled feature matrix.
- X (pd.DataFrame): Original feature matrix.
- y (pd.Series): Target variable.

Returns

- pd.Series: Feature importances sorted in descending order.
- RandomForestClassifier: Trained Random Forest model.

`eda.`**`rfe_selection`**`(`*X_scaled*, *y*, *X*, *rf*`)`

Recursive Feature Elimination.

Args

- X_scaled (np.ndarray): Scaled feature matrix.
- y (pd.Series): Target variable.

- X (pd.DataFrame): Original feature matrix.
- rf (RandomForestClassifier): Trained Random Forest model.

Returns

- pd.Series: Boolean mask indicating selected features.

eda.**rfecv_selection**(*X_scaled*, *y*, *X*, *rf*)

RFECV - RFE with cross-validation.

Args

- X_scaled (np.ndarray): Scaled feature matrix.
- y (pd.Series): Target variable.
- X (pd.DataFrame): Original feature matrix.
- rf (RandomForestClassifier): Trained Random Forest model.

Returns

- pd.Series: Boolean mask indicating selected features.

eda.**sfs_selection**(*X_scaled*, *y*, *X*, *rf*)

Sequential Feature Selector.

Args

- X_scaled (np.ndarray): Scaled feature matrix.
- y (pd.Series): Target variable.
- X (pd.DataFrame): Original feature matrix.
- rf (RandomForestClassifier): Trained Random Forest model.

Returns

- pd.Series: Boolean mask indicating selected features.

CHAPTER

FIFTEEN

LABEL REAL DATASET

`add_current_uc.`**`apply_uc`**`(`*row*`)`

Assign a Use Case (UC) label to a row based on its timestamp.

Args

- row (pd.Series): A row with a 'timestamp' field.

Returns

- str: The UC label ('uc1' to 'uc6').

`add_current_uc.`**`compare_intervals`**`(`*row*, *interval*`)`

Compare a timestamp from a row with a given time interval.

Args

- row (pd.Series): Row from the DataFrame with a 'timestamp' column.
- interval (dict): Dictionary with "from" and "to" datetime strings.

Returns

- bool: True if timestamp is within interval, else False.

`add_current_uc.`**`label_realnetwork_csv`**`(`*input_path='../datasets/real_network_data_before_labeling.csv'*, *output_path='../datasets/real_network_data_after_labeling.csv'*`)`

Load the CSV, assign UC labels to each row, and save the updated file.

Args

- input_path (str): Path to the input CSV file.
- output_path (str): Path to save the labeled CSV file.

Returns

None

PREPROCESS DATA FOR LSTM

Module for preparing LSTM input data from preprocessed features. Includes functionality for loading feature arrays, creating sequences, splitting the dataset, and saving the output for training and testing.

`lstm_preprocessing.`**`create_sequences`**`(`*X*, *y*, *seq_len*`)`

Creates sequences of input data for LSTM using a sliding window.

Args

- **`X`** (*`np.ndarray`*) – Input data (features)
- **`y`** (*`np.ndarray`*) – Target values (classes)
- **`seq_len`** (*`int`*) – Length of the sequence for LSTM

Returns

- tuple: (X_seq, y_seq) as ndarray

`lstm_preprocessing.`**`load_data`**`(`*X_path*, *Y_path*, *scaler_path*, *features_path*, *uc_map_path*`)`

Loads the preprocessed data from specified paths.

Args

- X_path (str): Path to the input features (X)
- Y_path (str): Path to the target labels (y)
- scaler_path (str): Path to the scaler object
- features_path (str): Path to the selected features JSON file
- uc_map_path (str): Path to the UC map JSON file

Returns

- tuple: (X, y, scaler, selected_features, uc_map)

`lstm_preprocessing.`**`split_and_save_data`**`(`*X_seq*, *y_seq*, *output_dir='preprocessed_data'*`)`

Splits the dataset into training and testing sets and saves them to disk.

Args

- X_seq (np.ndarray): Input features in sequence format
- y_seq (np.ndarray): Target labels in sequence format
- output_dir (str): Directory to save the split data

Returns

None

CHAPTER SEVENTEEN

SIMULATE A RUNNING NETWORK

`running_network.init_log()`

Initializes the log file and directory if needed.

Args
None

Returns
None

`running_network.run_simulation(`*script_name*`)`

Runs the selected UC simulation and logs its outcome.

Args

- script_name (str): Name of the UC script to run.

Returns
None

CHAPTER

EIGHTEEN

MAIN ORCHESTRATOR

class network_watcher.**AttentionLayer**(**args*, ***kwargs*)

Bases: `Layer`

Custom attention layer for LSTM model. This layer computes the attention weights and applies them to the input sequence.

Args

- Layer (tf.keras.layers.Layer): Base class for all layers in Keras.

Returns
None

build(*input_shape*)

Create the attention weights and bias.

Args

- input_shape (tuple): Shape of the input tensor.

Returns
None

call(*x*)

Calculate the attention weights and apply them to the input sequence.

Args

- x (tensor): Input tensor of shape (batch_size, sequence_length, features).

Returns

- tensor: Output tensor of shape (batch_size, features).

compute_output_shape(*input_shape*)

Compute the output shape of the layer.

Args

- input_shape (tuple): Shape of the input tensor.

Returns

- tuple: Shape of the output tensor.

network_watcher.**clean_old_models**(*directory='/app/data/Model'*, *keep_last_n=7*, *pattern='Model_bn_*.keras'*)

Keep only the last *keep_last_n* saved model files and delete older ones.

Args

- directory (str): Directory containing the model files.
- keep_last_n (int): Number of recent models to keep.
- pattern (str): Pattern to match model files.

Returns
None

`network_watcher.`**`load_last_sequence`**`(`*csv_path*, *selected_features*, *sequence_length=60*`)`

Load the last sequence of records from CSV, ensuring all required features and labels are present.

Args

- csv_path (str): Path to the CSV file.
- selected_features (list): List of features to select from the DataFrame.
- sequence_length (int): Length of the sequence to load.

Returns

- tuple: DataFrame with selected features and the correct labels.

Raises

- ValueError: If any of the selected features are missing in the DataFrame.

`network_watcher.`**`main_loop`**`(`*interval=1*, *prometheus_port=9000*`)`

Main loop that monitors UE activity, parses logs, updates Prometheus metrics, and fine-tunes the model in real-time.

Args

- interval (int): Time interval for monitoring and updating metrics.
- prometheus_port (int): Port for Prometheus metrics.

Returns
None

`network_watcher.`**`parse_amf`**`(`*lines*, *previous_state*`)`

Parse AMF log lines to extract UE registration and deregistration events, update UE states, and compute registration/session durations.

Args

- lines (list): List of log lines to parse.
- previous_state (dict): Previous state of UE details and durations.

Returns

- tuple: Updated UE details, new registration durations, and new session durations.

`network_watcher.`**`predict_current_uc`**`(`*latest_window_df*`)`

Predict the current use case (UC) using the loaded LSTM model based on the latest data window.

Args

- latest_window_df (pd.DataFrame): DataFrame containing the latest data window.

Returns

- tuple: Predicted UC class and confidence score.

`network_watcher.`**`remove_offset`**`()`

Remove the offset file used by Pygtail to start reading the log file from the beginning.

Args
None

Returns
None

`network_watcher.`**`run_main_notebook_with_backup`**`()`

Execute the main.ipynb notebook, log its execution, truncate CSV, and periodically create CSV backups.

Args
None

Returns
None

`network_watcher.`**`run_notebook_in_thread`**`()`

Run the main.ipynb notebook in a separate thread to avoid blocking the main loop.

Args
None

Returns
None

`network_watcher.`**`save_model_with_date`**`(`*model*, *path_prefix='/app/data/Model/Model_bn_'*`)`

Save the current model to disk with the current date as part of the filename.

Args

- model (tf.keras.Model): The model to save.
- path_prefix (str): Prefix for the filename.

Returns
None

`network_watcher.`**`truncate_running_data`**`(`*csv_path*, *keep_last_n=60*`)`

Truncate the CSV file to keep only the last *keep_last_n* records.

Args

- csv_path (str): Path to the CSV file.
- keep_last_n (int): Number of records to keep.

Returns
None