# find_anomalies

```
Main Module

 Program Execution Params:
 param --mode: Mode to run the algorithm. (Available: Train/Test)
 param --model: Filename to save/load the trained model.
 param --evaluate: Run evaluation test in order to propose missing thresholds.
                   If all thresholds are set by user, evaluation is not required.
                   In case that some thresholds are not set and evaluation is
                   disabled, non-set thresholds will be set with default values.
 param --thresholds_file: Filename, where thresholds are saved/loaded in JSON format.
 param --influx_host: Host IP of InfluxDB.
 param --influx_port: Port InfluxDB is listening.
 param --influx_user: InfluxDB username.
 param --influx_pass: InfluxDB user's password.
 param --influx_db: InfluxDB database, where metrics and anomalies are stored.
 param --influx_measurement: InfluxDB measurement in influx_db where anomalies are stored.


 Mode: Train
 Deep Learning model must be trained with a dataset containing normal records.
 Training data have been collected from both Prometheus and Amarisoft exporter.
 ----- Preprocessing -----
 Because of these two different systems there is a small difference in timestamps
 for these metrics. For this reason the training dataset is resampled every
 15 seconds, to synchronize records from these two systems.
 For training the following features have been selected: CPU percentage rate,
 RAM percentage rate, RX/TX CPU percentage rate, rate of transmitted/received
 bytes and rate of bytes downloaded/uploaded from 5G interface.
 These features will be used normalized using Min-Max Normalization. The min & max
 values that will be used for normalization will be saved in a JSON file. After
 resampling and normalization the dataset is splitted in sequences, based on a
 number of steps parameter. This parameters is set to 4, which means that for each
 bunch of 4 records, the first 3 will be used as historical data for the algorithm,
 and the fourth is the one that must be predicted.
 ----- Training -----
 For training the architecture of an autoencoder has been used. Model consisted of
 9 Biderectional LSTM layers and one Dense layer at the end. As activation function
 ReLu is used. SGD with Nesterov momentum has been used as optimizer, with learning
 rate equal to 0.01. A part of the trainset will be used as validation set. By default
 it is 10% of the training set.
 ----- Evaluation -----
 If evaluation is enabled, trained model will be loaded and used for testing in
 three different datasets. The first dataset has been collected during a CPU
 stress test attack. The second dataset has been collected during an iperf stress
 test attack. The third dataset is the training set. All these datasets will be
 normalized with saved normalization values during training. Using these datasets
 the RMSE error between the actual and predicted values will be calculated.
 Network features' thresholds, will be not affected from CPU stress dataset, and
 CPU features' thresholds, will be not affected from iperf stress dataset. The
 99th percentile of each feature's RMSE will be considered as threshold from
 anomaly detection algorithm. The 99th percentile has been selected compared with
 the max value, in order to avoid outliers, that may exist in these datasets.
 User-defined thresholds will not be updated from calculated RMSEs.


 Mode: Test
 Trained Model will be used in order to predict the next features' values in the
 time series. Because it is trained in normal traffic it will predict these values
 considering that the incoming traffic is normal. If the RMSE between the predicted
 and actual value is above a set threshold this record will be considered as anomaly.
 Algorithm fetches the last record from an InfluxDB every 15 seconds. For predicting
 the next value a sliding time window has been used, keeping only the last n records.
 By default this n is set to be equal to 30. Fetched records are been preprocessed
 and normalized before used by the model for predicting. After prediction the RMSEs
 between the actual and predicted values are calculated and compared with set thresholds,
 to identify if the record considers an anomaly or not. A possible cause of the anomaly
 is also saved in the database with the detected anomalies or a 'unknown cause' message
 if the cause cannot be identified.
```

## Modules

| | | | |
|---|---|---|---|
| argparse | math | ntpath | tensorflow |
| logging | numpy | pandas | time |

## Functions

**evaluate**(thresholds_file, cpu_testset, iperf_testset, trainset, time_window_threshold)

```
     Evaluate trained model. If user has not set all thresholds for anomalies, evaluation will also set
      the remaining thresholds. Evaluation uses a dataset, that contains a CPU stress test, an iperf
      stress test and predicting the data used for training. Thresholds are defined by calculating the
      RMSEs from actual values and taking the 99th percentile of these errors for each feature separately and overall.

      If user has set all thresholds when starting the program,these thresholds will be used.

      param thresholds_file: File, where user-
     defined thresholds are saved. This will updated if new thresholds are proposed.
      param cpu_testset: File containing dataset with CPU stress test.
```

```
 param iperf_testset: File containing dataset with iperf stress test.
 param trainset: File containing the dataset used for training.
 param time_window_threshold: Time window for keeping the last-
n records. In evaluation data are predicted in batches of n.

 return: None.
```

**seed**(...) method of numpy.random.mtrand.RandomState instance

```
seed(self, seed=None)

 Reseed a legacy MT19937 BitGenerator

 Notes
 -----
 This is a convenience, legacy function.

 The best practice is to **not** reseed a BitGenerator, rather to
 recreate a new one. This method is here for legacy reasons.
 This example demonstrates best practice.

 >>> from numpy.random import MT19937
 >>> from numpy.random import RandomState, SeedSequence
 >>> rs = RandomState(MT19937(SeedSequence(123456789)))
 # Later, you want to restart the stream
 >>> rs = RandomState(MT19937(SeedSequence(987654321)))
```

**sqrt**(x, /)

```
Return the square root of x.
```

**train**(train_dataset, n_steps, n_features, train_epochs=10, val_split=0.1, train_verbose=1, model_filename='model/5g_edge_autoencoder.h5')

```
Train DL model.

 param train_dataset: Dataset, that will be used for training model.
 param n_steps: How many previous steps in sequence will be used for training.
 param n_features: Number of features that will be used for training.
 param train_epochs: Epochs to train the model. (Default: 10)
 param val_split: Percentage of training dataset to use as validation dataset. (Values: 0.0 - 1.0) (Default: 0.1)

 param train_verbose: Show output of training. (Values: 0, 1, 2) (Default: 1)
 param model_filename: Filename to save trained model. (Default: model/5g_edge_autoencoder.h5)

 return: Trained model.
```

## Data

**cpu_testset** = 'data/cpu_attack.csv'
**data_prefix** = 'data/'
**iperf_testset** = 'data/iperf_attack.csv'
**model_prefix** = 'model/'
**plots_prefix** = 'plots/'
**stats_json** = 'data/normalization_stats.json'
**trainset** = 'data/normal.csv'

# influx_utils

[index](index)
[c:\users\thanasis\desktop\5g_anomaly_detection\influx_utils.py](c:\users\thanasis\desktop\5g_anomaly_detection\influx_utils.py)

```
InfluxDB Functions
```

## Modules

logging    numpy    time

## Functions

**checkDatabase**(client, db_name)

```
Checks if database, where metrics and detected anomalies will be stored, exists.
 If database does not exists, it will be created. Otherwise return with no action.

 param client: Influx client. Client must be initialized and connection with Influx must be established.
 param db_name: Database name, where metrics and detected anomalies will be stored.

 return: None
```

**getLastRecords**(client, queries, measurements)

```
Fetches last inserted record for each metric. Executes a list of quesries.
 Each executed query fetches the last inserted record for one measurement.

 param client: InfluxDB client object.
 param queries: List of influx queries, that will be executed.
 param mesaurements: list of measurements, that match the queries.

 return: data, timestamp
 data: A python dict containing all fetched data. Dict's key is measurement name and value the 'value' column of Influx DB.
 timestamp: A common timestamp for all measurements, to synchronize some of them that are inserted with a delay compared with the rest.
```

**initializeConnection**(influx_host, influx_port, influx_user, influx_pass)

```
Initialize connection with Influx DB.

 IMPORTANT: User must have rights to create a new database.
 If user cannot create new database, detected anomalies will not be inserted in database.
 Database, can also created manually.

 param influx_host: Host's IP, where Influx is running.
 param influx_port: Host's port, where Influx service is listening.
 param influx_user: Username for Influx connection.
 param influx_pass: User's password.

 return: Influx client object
```

**insertAnomalies**(client, influx_measurement, timestamps, anomaly_np, tag_msg)

```
Inserts the detcted anomalies in the defined measurement in Influx DB.

 param client: InfluxDB client object.
 param influx_measurement: Measurement, where the anomalies will be stored. It is defined as parameter when the main program starts.
 param timestamps: Timestamp of detected anomaly.
 param anomaly_np: A numpy array, that contains metrics' values for detected anomaly.
 param tag_msg: String, with a possible cause of detected anomalies. If there is no clear cause, its value will be 'unknown cause'.

 return: None
```

# metrics_formatter

```
Format retrieved metrics from InfluxDB
```

## Modules

logging    pandas

## Functions

**format5gCpuPercentage**(json_array, last_record)

```
     Receives percentage usage of RX/TX CPU. Calculate the rate of usage
      between json_array and last_record.

      param json_array: JSON Array with percentage of RX/TX CPU.
      param last_record: previous retrieved record from InfluxDB.

      return: percentage of RX/TX CPU and rate of usage.
```

**format5gNetworkBytes**(json_array, last_record)

```
     Receives bits of 5G interface.

      param json_array: JSON Array with number of bites for 5G interface.
      param last_record: previous retrieved record from InfluxDB.

      return: Total number of bytes and rate of bytes for 5G interface.
```

**formatCpuSecondsTotal**(json_array, last_record)

```
     Receives CPU seconds total. Filter CPU seconds, keeping only 'user' mode. For both of the 2
      different CPUs calculate percentage usage using the time difference from the last_record param
      and the difference of seconds in 'user' mode. Also, based on last_record's percentage calculate
      the rate of the percentage for each CPU separately, and aggregated. Finally, return json_array
      in order to be used as last_record for the next retrieved metrics.

      param json_array: JSON Array with total cpu seconds in node.
      param last_record: previous retrieved record from InfluxDB.

      return: json_array, features_core, features
      json_array: input param
      features_core: formatted percentage & rate for mode user for both cpu 0 and 1
      features: aggregated formatted percentage & rate for mode user
```

**formatMemoryFreeBytes**(json_array, last_record)

```
     Receives free bytes of available RAM. Total size of RAM is 8228077568.0 bytes.
      Substracting from total size calculate the the used RAM, percentage of RAM used.
      Using last_record calculate the rate of RAM usage.

      param json_array: JSON Array with number of free RAM bytes.
      param last_record: previous retrieved record from InfluxDB.

      return: Used RAM, percentage of used RAM and rate of RAM usage.
```

**formatNetworkBytes**(json_array, last_record)

```
     Receives bytes of all network interfaces. Filter them, and keep 'enp1s0', 'enp0s20u1' and 'ppp0'
      interfaces. It calculates using last_record bytes and bytes rate for each interface separately.

      Also it calculates the total bytes and total rate aggregated from all three interfaces.

      param json_array: JSON Array with number of bytes in each interface.
      param last_record: previous retrieved record from InfluxDB.

      return: features_interfaces, features.
      features_interfaces: number of bytes and rate of bytes for each interface separately.
      features: Total number of bytes and aggregated rate of bytes from the three used interfaces.
```

# utils

index
c:\users\thanasis\desktop\5g_anomaly_detection\utils.py

Helper Functions

## Modules

| | | | |
|---|---|---|---|
| json | math | pandas | time |
| logging | numpy | matplotlib.pyplot | |

## Functions

**array**(...)

```
array(object, dtype=None, *, copy=True, order='K', subok=False, ndmin=0)

 Create an array.

 Parameters
 ----------
 object : array_like
     An array, any object exposing the array interface, an object whose
     __array__ method returns an array, or any (nested) sequence.
 dtype : data-type, optional
     The desired data-type for the array.  If not given, then the type will
     be determined as the minimum type required to hold the objects in the
     sequence.
 copy : bool, optional
     If true (default), then the object is copied.  Otherwise, a copy will
     only be made if __array__ returns a copy, if obj is a nested sequence,
     or if a copy is needed to satisfy any of the other requirements
     (`dtype`, `order`, etc.).
 order : {'K', 'A', 'C', 'F'}, optional
     Specify the memory layout of the array. If object is not an array, the
     newly created array will be in C order (row major) unless 'F' is
     specified, in which case it will be in Fortran order (column major).
     If object is an array the following holds.

     ===== ========= ===================================================
     order  no copy                     copy=True
     ===== ========= ===================================================
     'K'   unchanged F & C order preserved, otherwise most similar order
     'A'   unchanged F order if input is F and not C, otherwise C order
     'C'   C order   C order
     'F'   F order   F order
     ===== ========= ===================================================

     When ``copy=False`` and a copy is made for other reasons, the result is
     the same as if ``copy=True``, with some exceptions for `A`, see the
     Notes section. The default order is 'K'.
 subok : bool, optional
     If True, then sub-classes will be passed-through, otherwise
     the returned array will be forced to be a base-class array (default).
 ndmin : int, optional
     Specifies the minimum number of dimensions that the resulting
     array should have.  Ones will be pre-pended to the shape as
     needed to meet this requirement.

 Returns
 -------
 out : ndarray
     An array object satisfying the specified requirements.

 See Also
 --------
 empty_like : Return an empty array with shape and type of input.
 ones_like : Return an array of ones with shape and type of input.
 zeros_like : Return an array of zeros with shape and type of input.
 full_like : Return a new array with shape of input filled with value.
 empty : Return a new uninitialized array.
 ones : Return a new array setting values to one.
 zeros : Return a new array setting values to zero.
 full : Return a new array of given shape filled with value.


 Notes
 -----
 When order is 'A' and `object` is an array in neither 'C' nor 'F' order,
 and a copy is forced by a change in dtype, then the order of the result is
 not necessarily 'C' as expected. This is likely a bug.

 Examples
 --------
 >>> np.array([1, 2, 3])
array([1, 2, 3])

 Upcasting:

 >>> np.array([1, 2, 3.0])
array([ 1.,  2.,  3.])

 More than one dimension:
```

```
 >>> np.array([[1, 2], [3, 4]])
array([[1, 2],
       [3, 4]])

 Minimum dimensions 2:

 >>> np.array([1, 2, 3], ndmin=2)
array([[1, 2, 3]])

 Type provided:

 >>> np.array([1, 2, 3], dtype=complex)
array([ 1.+0.j,  2.+0.j,  3.+0.j])

 Data-type consisting of more than one element:

 >>> x = np.array([(1,2),(3,4)],dtype=[('a','<i4'),('b','<i4')])
 >>> x['a']
array([1, 3])

 Creating an array from sub-classes:

 >>> np.array(np.mat('1 2; 3 4'))
array([[1, 2],
       [3, 4]])

 >>> np.array(np.mat('1 2; 3 4'), subok=True)
 matrix([[1, 2],
         [3, 4]])
```

**convertNumpyToPandas**(np_arr)

```
Create a Pandas DataFrame object from a numpy array.
 First column of numpy array will be used as index.

 param np_arr: numpy array to be converted in Pandas DataFrame

 return: The ccreated Pandas DataFrame
```

**createEmptyMetricsArray**(col_num)

```
Creates an empty numpy array with one row and col_num columns, filled with zeros.

 param col_num: Number of columns.

 return: An array with size (1 x col_num) filled with zeros.
```

**getAnomalyColumns**()

```
Creates a list of columns from dataset, that will be saved in InfluxDB when an anomaly is detected

 return: List of columns from dataset
```

**getColumnNames**()

```
Creates a Python dict object, where dict's keys are columns names and dict's values are columns' types.

 return: dict with columns names and types.
```

**getFeatures**()

```
Creates a list of columns ftom dataset, that will be used for training & testing

 return: List of columns from dataset
```

**loadDataset**(dataset)

```
Loads a dataset into a Pandas DataFrame. Dataset file must be in .csv format.
 It must be separated with commas (,), has a header with column names.
 Also, it must has a column named time, which will have timestamp values.
 Column time is parsed as date and set as the index of the generated DataFrame.

 param dataset: File where the dataset is saved.

 return: Dataframe generated from dataset param
```

**loadDictJson**(filename)

```
Loads a Python dict object from a given file. Given file must be in JSON format.

 param filename: Filename(included path) that will be converted in dict.

 return: The loaded Python dict object.
```

**normalizeFeature**(df, feature, min, max)

```
Min-Max Normalization. Creates a new Pandas Series with normalized values of another Pandas Series.
 Minimum and Maximum values that will be used for normalization must be provided.

 param df: Pandas DataFrame object
 param feature: Column of df param, that will be normalized
 param min: Minimum value that will be used for normalization
 param max: maximum value that will be used for normalization

 return: Pandas Series, where its values is the normalized ones from feature param according to min and max params
```

**plotAccLoss**(history, plots_prefix)

```
Plots Accuracy & Loss metrics in one plot together
 and save it to .eps, .png and .tiff file

 param history: history of training
 param plots_prefix: data path, where plots are saved
 return: None
```

**plotMetric**(history, metric, plots_prefix)

```
Plots metric and save it to .eps and .png files

 param history: history of training
 param metric: metric name
 param plots_prefix: data path, where plots are saved

 return: None
```

**printPredictionErrors**(y_actual, y_predict)

```
Calculate RMSE for predicted features.
 Creates a dict with calculate RMSE for all features combined, and for each one separately.

 param y_actual: Real values of a sequence.
 param y_predict: Predicted values of a sequence.

 return: Dict with calculated RMSEs.
```

**saveDictJson**(dict, filename)

```
Dumps a Python dict object to a given file. Created file will be in JSON format.

 param dict: Python dict object.
 param filename: Filename(included path) where the dict will be saved.

 return: None
```

**saveNormalizationStats**(df, cols_to_normalize)

```
Dump a Python dict, that contains min & max value for each feature, into a JSON file.
 These min & max values will be used to normalize required features.

 param df: Pandas DataFrame object
 param cols_to_normalize: List of columns to normalize from df param

 return: A Python dict object, that contains min & max values for each feature that will be normalized
```

**split_sequences**(sequences, n_steps)

```
Split a multivariate sequence into samples. (n-1) steps will be considered as previous records in the sequence,
 and last step will be the current value of the sequence. The first (n-1) will be used from DL model in order to
 predict the n-th value of the sequence.

 param sequnces: Sequence that will be splited.
 param n_steps: Number of steps, that will be used for produced splitted sequences.

 return: array(X),array(y)
array(X): Array containing the previous values of the sequence
array(y): Array containing the current value of the sequence
```

**sqrt**(x, /)

```
Return the square root of x.
```