tegdet: An extensible Python Library for Anomaly Detection using Time-Evolving Graphs

Simona Bernardi, José Merseguer

Universidad de Zaragoza, Spain

Raúl Javierre Hiberus, Spain

Abstract

This paper presents a new Python library for anomaly detection in unsupervised learning approaches. The input for the library is a univariate time series representing observations of a given phenomenon. Then, it can identify anomalous epochs, i.e., time intervals where the observations are above a given percentile of a baseline distribution, defined by a dissimilarity metric. Using time-evolving graphs for the anomaly detection, the library leverages valuable information given by the inter-dependencies among data. Currently, the library implements 28 different dissimilarity metrics, and it has been designed to be easily extended with new ones. Through an API, the library exposes a complete functionality to carry out the anomaly detection. Summarizing, to the best of our knowledge, this library is the only one publicly available, that based on dynamic graphs, can be extended with other state-of-the-art anomaly detection techniques. Our experimentation shows promising results regarding the execution times of the algorithms and the accuracy of the implemented techniques. Additionally, the paper provides guidelines for setting the parameters of the detectors to improve their performance and prediction accuracy.

1. Introduction

Anomaly detection is a research field initiated in the 20th century by the statistics community, being the work of Grubbs (1969), among others, a reference. The computer science community started to research and apply anomaly detection techniques in the 80s of the previous century. These techniques are currently applied in many different and heterogeneous fields, such as network intrusion detection, malware detection, fraud detection, data center monitoring, social networks or social media research. Undoubtedly, for an industrial adoption of these techniques, also for researchers to leverage them, it is mandatory the existence of software packages, whose implementation favors ease of use. Moreover, considering the myriad of existing anomaly detection techniques, software extensibility is another important property. In this context, extensibility refers to the capability of easily adding new anomaly detection techniques to the software package, and not needing to fully learn the package implementation.

Since many are the fields where spotting anomalies is a need, then many are the anomaly detection techniques. Some of them can identify outliers in unstructured collections of multi-dimensional data points. However, in many domains, such as finance, security, communication media or health care, data exhibit inter-dependencies, which provide valuable information to detect outlier data. Hence, graphs have been prevalently used to represent structural/relational information among data in many domains, e.g. Aggarwal, Zhao, and Yu (2011); Wu, Pan, Zhu, Zhang, and Yu (2018), then raising the graph anomaly detection problem. Akoglu, Tong, and Koutra (2015) identified four reasons that highlight the need of graphs for anomaly

detection: a) inter-dependent nature of the data, b) powerful representation for such inter-dependencies, c) relational nature of problem domains and d) robust machinery, for graphs to serve as adversarial tools also. More specifically, *dynamic* or *time-evolving graphs*, i.e., sequences of static graphs, have been extensively used in the literature for detecting outliers in time series. The book of Gupta, Gao, Aggarwal, and Han (2014) surveys many anomaly detection techniques in this field, while Febrinanto, Xia, Moore, Thapa, and Aggarwal (2022) expose current challenges in the dynamic evolution of graph data.

This paper aligns with definitions given by Akoglu et al. (2015), which are based on the outlier definition given by Hawkins (1980): "an observation that differs so much from other observations as to arouse suspicion that it was generated by a different mechanism". Then, the dynamic graph anomaly detection problem is defined as "given a sequence of (plain/attributed) graphs, find (i) the timestamps that correspond to a change or event, as well as (ii) the top-k nodes, edges, or parts of the graphs that contribute most to the change". Currently, our work deals with the first part of the definition, while the finding of "top parts of the graphs" will be the subject of future versions.

Regarding publicly available implementations of dynamic graph anomaly detection techniques, the recent survey by Ma, Wu, Xue, Yang, Zhou, Sheng, Xiong, and Akoglu (2021) identifies eight works and their corresponding repositories. In the following, we review these works. SedanSpot (Eswaran and Faloutsos 2018) implements in C++ (Stroustrup 2013) an algorithm to detect anomalies from an edge stream in near real-time, where anomalies are edges connecting sparsely-connected parts of the graph (bridge edges) and possible occurring during intense bursts of activity. AnomalyDAE (Fan, Zhang, and Li 2020) is a Python (van Rossum 2007) code implementing a deep learning anomaly detector, that aims at finding nodes whose patterns deviate significantly from the majority of reference nodes. ChangeDAR (Hooi, Akoglu, Eswaran, Pandey, Jereminov, Pileggi, and Faloutsos 2018), also developed in Python, detects change points using sensors placed on nodes or edges. It can be applied to detecting electrical failures, traffic accidents or other events in road traffic graphs. AnomRank (Yoon, Hooi, Shin, and Faloutsos 2020) implements a C++ algorithm that detects sudden weight changes along an edge and also sudden structural changes to the graph. A GitHub repository implements DAGMM Zong, Song, Min, Cheng, Lumezanu, Cho, and Chen (2018), as a set of Python scripts. DAGMM is an unsupervised anomaly detection technique that uses a deep autoencoder to generate a low-dimensional representation and reconstruction error for each input data point, which is fed into a Gaussian Mixture Model. Then, it optimizes the parameters and the model jointly. F-FADE (Chang, Li, Sosic, Afifi, Schweighauser, and Leskovec 2021) is an anomaly detection technique, implemented in Python, that applies to edge streams. It develops a technique, that requires constant memory, to efficiently model time-evolving distributions between node-pairs. MIDAS (Bhatia, Hooi, Yoon, Shin, and Faloutsos 2020) is an online technique focussed on detecting microcluster anomalies, groups of suspicious similar edges, instead of individually edges. It has been implemented in C++. DeepSphere (Teng, Yan, Ertugrul, and Lin 2018) develops and implements in Python an unsupervised algorithm to address case-level and nested level anomaly detection. It does not require outlier-free data as input and can reconstruct normal patterns. Besides these eight works, we have also identified many other interesting works, such as Li, Han, Cheng, Su, Wang, Zhang, and Pan (2019); Liu, Pan, Wang, Xiong, Wang, Chen, and Lee (2021); Zheng, Li, Li, Li, and Gao (2019). However, we have not found available implementations of these

¹https://github.com/danieltan07/dagmm

works, which is the subject of this paper.

This paper presents a novel Python library for anomaly detection, based on dynamic graphs. The input of the approach, also of the library, must be a univariate time series representing observations of a given phenomenon. The output identifies *anomalous epochs*, i.e., time intervals where the observations are above a given percentile of a distribution, which is defined by a given dissimilarity metric. Currently, the library implements 28 different dissimilarity metrics, and it has been designed to be easily extended. Therefore, a prominent feature of the library is the ease for users to introduce new anomaly detection techniques.

Unlike our work, most of the previously reviewed software presents the implementation of a graph-based anomaly detection technique, which has been usually developed by the very same authors, to solve a specific domain problem. These implementations aim at proving the efficiency and correctness of the technique. Thus, none of these works are specifically interested on defining a Python library exposing a proper API, as a mean to offer the functionalities of the anomaly detection for being integrated into other packages. Another salient difference is that there is no concern in these works for developing a software design that supports the extensibility of the package, with new anomaly detection techniques proposed by other authors. On the contrary, our library is completely focussed on being a reference, in Python, for integrating implementations of state-of-the-art anomaly detection techniques, based on dynamic graphs. Summarizing, our library is ready to be used by other software implementations and ready to be extended with other techniques. The only conditions for a technique, such as those developed in the reviewed works, to be integrated in the library are that: a) it accepts, as input, a univariate time series and b) it fits in an unsupervised learning approach, as described in Section 2.

Finally, our experimentation shows promising results regarding the execution times of the algorithms (i.e., time to build the prediction model and time to get the *anomalous epochs*) and the accuracy of the implemented anomaly detectors for the considered dataset. Additionally, the paper offers guidelines, for setting the parameters of the anomaly detectors, which improve the performance and prediction accuracy.

The rest of the paper is structured as follows. Section 2 presents the anomaly detection approach implemented by the library. Section 3 overviews the software package and the use of the library. Section 4 recalls an experiment to detail the use of the library. Section 5 assesses the quality of the library. Section 6 offers guidelines for the users to set the parameters of the library, so to get the most of it. Section 7 concludes the work. Appendices A, B and C offer necessary technical details for the correct use and extension of the library.

2. Models and algorithms

Time Evolving Graphs (TEGs, Shumway and Stoffer (2011)) are sequences of static graphs. TEGs have been successfully used for anomaly detection in unsupervised learning approaches (Akoglu et al. 2015). Figure 1 summarizes the unsupervised learning approach followed in this work. In such approaches, a prediction model is built first, from training datasets, which represent the behaviour under study. Then, this model is fed with testing datasets to detect outliers in the future behaviour. In our approach, input datasets, training and testing, are univariate time series, and they are used to create TEGs. The prediction model is obtained from the training TEGs, while the outliers detection uses the prediction model and the testing TEGs.

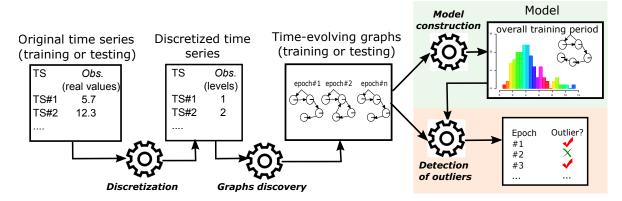


Figure 1: Unsupervised learning approach

The creation of TEGs is a two-step process: discretization and graphs discovery. First, the original time series (training or testing dataset) is discretized, which means to map real valued observations onto an ordered set of levels. Second, the discretized time series is used to create (discover) the TEGs. Concretely, by partitioning the overall time period of the series into equal sized consecutive time intervals, namely epochs. Therefore, each graph synthesizes observations of the original time series during an epoch. Hence, the granularity of the TEGs depends on two factors: the number of epochs and the level of discretization. The former defines the number of graphs, which is equal to the number of epochs. The latter affects the size of the graphs, in terms of the number of nodes and edges.

In a subsequent step, a prediction model is built from the training TEGs. The model is made of a global graph and a baseline distribution. The former is a graph synthesizing the TEGs, and therefore, the observations from the entire training dataset. The latter is the distribution of the dissimilarities between each graph of the training TEGs and the global graph. Since different dissimilarity metrics can be used to calculate such distribution, then, there are as many different anomaly detectors as dissimilarity metrics.

The anomaly detection encompasses two steps: the generation of testing TEGs and the detection of outliers. The latter computes the dissimilarities between each graph of the testing TEGs and the global graph of the prediction model. An epoch is then labeled as anomalous if the dissimilarity between its corresponding graph and the global graph is above the k^{th} percentile of the baseline distribution, where k is a parameter of the detector.

In the following, the four steps in Figure 1 are formalized: discretization, graphs discovery, model construction and detection of outliers.

2.1. Discretization

A univariate time series, of a real valued variable, represents an ordered set of observations (or data points, o) taken at regular time instants (timestamps, ts), e.g., every 30 minutes. Discretization means mapping real-valued observations into an ordered set of levels, that is, integer values, through a discretization function. Hence, the function depends on the observations and on the chosen number of levels, the level of discretization. The discretization function is defined prior to construct the prediction model, i.e., considering only the training dataset.

Discretization function. Let $\mathcal{T} = \{\langle ts_k, o_k \rangle\}_{k \in K \subset \mathbb{N}}$ be a univariate time series. We denote as:

$$m = \min_{k \in K} \{o_k\}, \ M = \max_{k \in K} \{o_k\}$$

the minimum and maximum data points, respectively.

The set of data points is partitioned into n intervals, level of discretization, of equal length $[m,M] = \bigcup_{i=0}^{n-1} [x_i,x_{i+1})$. Hence, the real value domain is partitioned into n+2 intervals:

$$\mathbb{R} = (\infty, m) \cup_{i=0}^{n-1} [x_i, x_{i+1}) \cup [M, \infty)$$

where $x_0 = m$ and $x_n = M$.

Then, the discretization function $\mathcal{D}: \mathbb{R} \to \mathbb{N}$ is defined as follows:

$$\mathcal{D}(x) = \begin{cases} i, & \text{if } x \in [x_i, x_{i+1}), \ i = 0, \dots, n-1 \\ n, & \text{if } x \in (\infty, m) \\ n+1, & \text{if } x \in [M, \infty) \end{cases}$$
 (1)

Observe that, n will not be assigned to observations in the training dataset. However, it is possible that it could be assigned to some observations in the testing dataset, during the detection phase.

2.2. Graphs discovery

Graph discovery is the second, and last, step in the generation of TEGs. This step actually produces a sequence of graphs $\mathcal{G} = \{G_i\}_{i \in J \subset \mathbb{N}}$ from a discretized time series $\mathcal{T} = \{G_i\}_{i \in J \subset \mathbb{N}}$ $\{\langle ts_k, level_k \rangle\}_{k \in K \subset \mathbb{N}}.$

 \mathcal{G} is generated considering *epochs*, i.e., consecutive time intervals of equal size s (number of observations per epoch), where s is an input parameter, see Algorithm 1. Let |K| = N be the total number of observations in \mathcal{T} , then the epochs are defined as follows:

$$epoch_j = [ts_{js}, ts_{(j+1)s}), \quad j \in J = \{0, \dots, |N/s| - 1\}$$

Algorithm 1: Graphs discovery

```
Data: \mathcal{T} = \{\langle ts_k, level_k \rangle\}_{k \in K} (discretized time series), s (num. of observations per
```

Result: $\mathcal{G} = \{G_j\}_{j \in J}$ (time-evolving graph)

- 1 $\mathcal{G} = \emptyset$;
- **2** foreach $j \in [0, ..., |N/s|-1]$ do
- $\mathcal{T}_j = \operatorname{extractTimeSeries}(\mathcal{T}, s, j)$; // time series in the interval [js, (j+1)s) $G_j = \operatorname{generateGraph}(\mathcal{T}_j)$;
- append $(G_j, \mathcal{G});$
- 6 end

Therefore, a graph $G_j \in \mathcal{G}$ synthetizes the observations in an epoch $j \in J$. Concretely, the nodes of G_j represent the discretized observations in j, i.e., the levels, whereas the edges represent transitions (changes) between levels.

Algorithm 2 sketches the generation of a graph G, from the portion of the discretized time series \mathcal{T} corresponding to an epoch of size s. The nodes $V = \{\langle l_i, w_i \rangle\}$ are pairs, where l_i is a level, taken from the observations in the considered epoch, and w_i is the number of occurrences of l_i in the epoch. V is ordered according to the levels l_i , and its size $|V| \leq s$. The edges E define an adjacency matrix, where each entry E_{rc} means the number of transition occurrences from level r to level c in the epoch. Thus, each graph is directed and weighted.

```
Algorithm 2: Graph generation
```

```
Data: \mathcal{T} = \{\langle ts_k, level_k \rangle\}_{k \in K'} (discretized time series with |K'| = s observations)

Result: G = (V, E) (weighted graph: weights are associated to both nodes and edges)

1 levels = extractLevels(\mathcal{T});  // levels is an ordered multiset of levels

// V is an ordered set of levels with their frequencies

2 V = \text{setNodes(levels, s)};

3 E = \text{initializeAdjacencyMatrix}(|V|);  // E is a |V| \times |V| matrix with 0-entries

4 foreach k \in [0, s-2] do

5 | r = \text{getIndex(levels, k)};

6 | c = \text{getIndex(levels, k+1)};

7 | E_{r,c} = E_{r,c} + 1;

8 end
```

2.3. Model construction

A prediction model $\mathcal{M}(G, D_{\mu})$ can be built using $\mathcal{G} = \{G_j\}_{j \in J \subset \mathbb{N}}$, a set of TEGs generated from a discretized training dataset. G is a global graph synthesizing the graphs in \mathcal{G} , and D_{μ} is a baseline distribution of the dissimilarities between each graph G_j and the global graph G. Since different dissimilarity metrics μ can be used to calculate this distribution, then there are just as many prediction models and, therefore, as many different anomaly detectors.

Global graph

A global graph is defined as the sum of all the graphs in \mathcal{G} :

$$G \stackrel{def}{=} \sum_{j=1}^{|J|} G_i = G_1 + G_2 + \dots + G_{|J|}$$
 (2)

where the *sum* binary operator is defined as follows.

Sum of graphs Let $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$ be two graphs, where $V_i = L_i \times \mathbb{N}(i = 1, 2)$ includes the set of pairs $\langle l, w \rangle$, ordered according to the levels $l \in L_i$, and $E_i(i = 1, 2)$ is the adjacency matrix representing the graph edges and their weights.

The sum of G_1 and G_2 , denoted as $G_1 + G_2$, is a graph G(V, E) where the set of nodes $V = (L_1 \cup L_2) \times \mathbb{N}$ is:

$$V = \{\langle l, w_1 + w_2 \rangle | \langle l, w_1 \rangle \in V_1 \land \langle l, w_2 \rangle \in V_2 \} \cup V_1 \setminus V_2 \cup V_2 \setminus V_1$$

The adjacency matrix $E = [e_{ij}]$ of G is a matrix of dimension $|L_1 \cup L_2| \times |L_1 \cup L_2|$, with the following entries:

$$e_{ij} = \begin{cases} x_{ij} + y_{ij} & \text{if } i, j \in L_1 \cap L_2 \\ x_{ij} & \text{if } i, j \in L_1 \ \land \ (i \not\in L_2 \lor j \not\in L_2) \\ y_{ij} & \text{if } i, j \in L_2 \ \land \ (i \not\in L_1 \lor j \not\in L_1) \end{cases}$$

where $E_1 = [x_{ij}]$ and $E_2 = [y_{ij}]$.

Baseline distribution

Let \mathcal{G} be a sequence of graphs, G its global graph, as defined in Equation 2, and $\mu: \mathcal{G} \times \{G\} \to \mathbb{R}_0^+$ a dissimilarity metric. Then, the baseline distribution is a function $D_{\mu}: J \to \mathbb{R}_0^+$ defined as follows:

$$D_{\mu}(j) = \mu(G_j, G) \tag{3}$$

There exists many metrics that can be used to compute dissimilarities between two graphs (Akoglu *et al.* 2015; Cha 2007). The **tegdet** library currently implements 28 different metrics, which are detailed in Appendix A.

Prior to compute a dissimilarity between G and a given G_j , it is mandatory to resize G_j . This means to expand the set of nodes and the incidence matrix of G_j to obtain a new graph with the same ordered set of nodes as in G (Algorithm 3). In the expansion, the frequencies associated to the new matrix entries are set to a wildcard value, so to differentiate them from those with zero values, which indicate the absence of an edge but the presence of a node. Such wildcard, will be treated differently, according to the concrete metric μ .

Algorithm 3: Graph resizing

```
Data: G_1 = (V_1, E_1), where V_1 = L_1 \times W_1 (graph to be resized);
  L (ordered set of levels of G)
1 wildcard = -1;
2 foreach level \in L do
     if level \notin L_1 then
3
         // Expand the graph
         position = getPosition(level);
         insertNewNode(V_1, position, \langle level, wildcard \rangle);
         insertNewColumn(E_1, position, wildcard);
6
         insertNewRow(E_1, position, wildcard);
7
      end
8
9 end
```

In the following, two different examples of baseline distributions, D_{μ} , are presented. The first one uses the Hamming (1950) metric, which is based on the structure of the graphs. The second one uses the Jeffreys (1961) metric, which is based on the edge relative frequencies.

Hamming dissimilarity distribution Formally, let $A = [a_{ij}]$ and $B = [b_{ij}]$ be the $n \times n$ adjacency matrices of the graphs to be compared, G_A and G_B respectively, where the wildcard entries are set to -1 value.

The Hamming dissimilarity distribution is defined as follows:

$$\mu(G_A, G_B) = 1 - \frac{\sum_{i=1}^n \sum_{j=1}^n \chi(a_{ij}, b_{ij})}{n^2}$$
(4)

where χ is the following indicator function:

$$\chi(x,y) = \begin{cases} 1 & \text{if } (x > 0 \land y > 0) \lor (x = 0 \land y = 0) \\ 0 & \text{otherwise} \end{cases}$$

Jeffreys dissimilarity distribution Formally, let $A = [a_{ij}]$ and $B = [b_{ij}]$ be the $n \times n$ adjacency matrices of the graphs to be compared, G_A and G_B respectively, where wildcard entries are set to zero value. Let $A' = [a'_{ij}]$ and $B' = [b'_{ij}]$ be the matrices obtained from A and B, respectively, where the entries represent relative frequencies:

$$a'_{ij} = \frac{a_{ij}}{(\sum_{k=1}^{n} \sum_{l=1}^{n} a_{kl})}, \ b'_{ij} = \frac{b_{ij}}{(\sum_{k=1}^{n} \sum_{l=1}^{n} b_{kl})}.$$

The Jeffreys dissimilarity distribution is defined as follows:

$$\mu(G_A, G_B) = \sum_{i=1}^{n} \sum_{j=1}^{n} (a'_{ij} - b'_{ij}) \log \frac{a'_{ij}}{b'_{ij}}$$
 (5)

2.4. Detection of outliers

Prediction models $\mathcal{M}(G, D_{\mu})$ are used to detect outliers in *testing* datasets. A *testing* dataset \mathcal{T}' is made of an ordered set of observations, accounting for the same phenomenon as the *training* dataset \mathcal{T} used to build the model. Such observations should be taken with the very same time granularity as for the training dataset.

The anomaly detection phase implies the generation of TEGs from \mathcal{T}' as described in sub-Sections 2.1 and 2.2. Hence, the discretization of \mathcal{T}' is carried out using the function (1), which is defined over the training dataset \mathcal{T} , whereas the graph discovery is performed considering the very same number of observations per epoch as in the model building phase.

Therefore, let $\mathcal{G}' = \{G'_j\}_{j \in J' \subset \mathbb{N}}$ be a set of TEGs obtained from the testing dataset \mathcal{T}' , then:

The epoch $j \in J'$ is **anomalous** if the dissimilarity between G'_j and the global graph G, computed using the metric μ , is above the $100-\alpha$ percentile of the baseline distribution D_{μ} .

Observe that $\alpha \in (0,100)$ represents the significance level and it is an input parameter. Algorithm 4 detects outlier epochs, given a prediction model, the significance level and the sequence of graphs generated from the training dataset.

Algorithm 4: Detection of outliers

```
Data: \mathcal{M}(G, D_{\mu}) (prediction model), \alpha (significance level), \mathcal{G}' = \{G'_j\}_{j \in J' \subset \mathbb{N}} sequence of graphs from \mathcal{T}'

1 thresold = \operatorname{percentile}(D_{\mu}, 100 - \alpha);

2 outlier = \emptyset;

3 foreach j \in J' do

4 | if \mu(G'_j, G) > thresold then

5 | outlier = outlier \cup \{j\};

6 | end

7 end
```

3. Implementation and use of the library

The **tegdet** library has been implemented in Python language (van Rossum 2007). As shown in Figure 2, the library uses three other different Python libraries and it is made of two packages:

- teg: It is the main package. It defines the API for the users of the library.
- graph comparison: It is responsible for creating graphs and computing the dissimilarity between two graphs, according to a given metric.

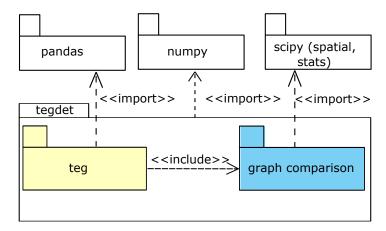


Figure 2: Overview of the **tegdet** library

Each package of the library is made up of a set of Python classes². Figure 3 depicts all the classes in the library, different colours are used to map classes to packages, according to Figure 2. In particular, the methods of the TEGDetector class conform the API of the tegdet library, they are described in Table 4, Appendix C.1. The methods of the API are for the users of the library to construct the prediction model and carry out the anomaly detection. In the following, both scenarios are described.

²A detailed description of the attributes and methods of these classes can be found in: https://github.com/DiasporeUnizar/TEG.

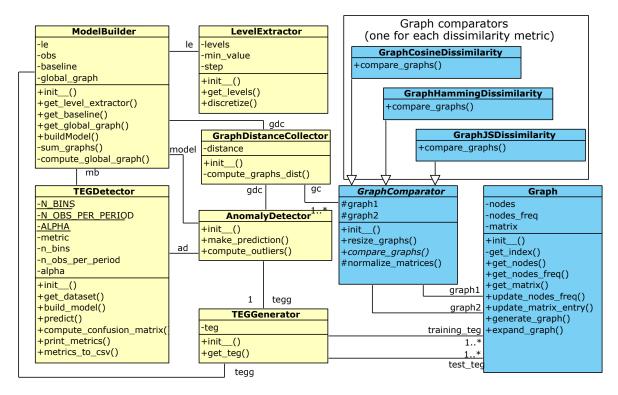


Figure 3: Class diagram of the **tegdet**

3.1. Construction of the prediction model

Figure 4 shows the interaction of the **tegdet** classes to carry out the construction of a *prediction model*. The *userScript* represents a Python script written by a user of the library. It must sequentially call three API methods. First, the init method sets some parameters needed to create the *training* TEGs and the prediction model. These parameters are detailed in Table 5 (Appendix C.1). In Figure 4, only the metric parameter is set since default values are left for the remaining parameters. Next, the get_dataset method loads the *training* dataset, as a pandas DataFrame. The third call, build_model, actually generates the prediction model, as follows.

Initially, instances of the ModelBuilder and LevelExtractor classes are created. Then, the build_model call to the ModelBuilder instance: a) produces a discretized dataset (discretize call), according to Function 1; b) generates a sequence of training graphs (generation of an instance of TEGGenerator class and first loop fragment), as described in Algorithm 1; c) obtains the global graph (compute_global_graph call), according to Equation 2; and d) computes the dissimilarities between each graph of the sequence and the global graph (generation of an instance of GraphDistanceCollector and last loop fragment), following Equation 3 and Algorithm 3. Finally, the results, that is the model and the time required to build it, are returned to the userScript.

It is worth to observe that, although this scenario shows an instance of the class GraphComparator, within the last loop fragment, in general the computation of graphs dissimilarities is carried out by a sub-class of GraphComparator (cfr. Figure 3, sub-classes of GraphComparator), in fact the sub-class corresponding to the metric to be computed.

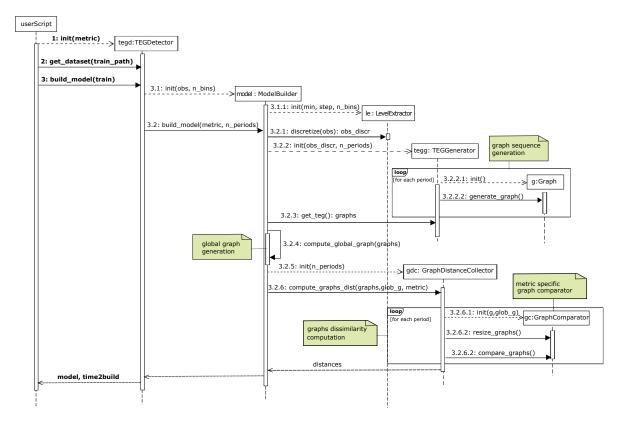


Figure 4: Model construction

3.2. Anomaly detection

Figure 5 shows the interaction of the **tegdet** classes to carry out the anomaly detection. First, observe that instances of TEGDetector, ModelBuilder and LevelExtractor have been already created in the model phase. The *userScript* starts the interaction by calling get_dataset to load the *testing* dataset as a pandas DataFrame. Then, the predict call triggers three sub-steps: the creation of an instance of AnomalyDetector, make_prediction and compute_outliers, which evolve as follows.

Predictions are carried out by: a) retrieving the discretization levels from the ModelBuilder and creating the discretized testing dataset (discretize call), according to Function 1; b) generating the corresponding sequence of testing graphs, as described in Algorithm 1; and c) computing the graph dissimilarities, following Equation 3 and Algorithm 3. Dissimilarities are computed using the global graph, retrieved from the ModelBuilder, and the metric selected in the first step. Then, the computation of outliers (Algorithm 4), i.e., the detection of anomalous epochs, is carried out considering the predictions and the baseline distribution, which is also retrieved from the ModelBuilder.

Next, the scenario computes a confusion matrix (compute_confusion_matrix call), that can be used to assess the prediction capabilities of the anomaly detector. In particular, its entries summarize the results of the anomaly detection as follows:

- True positives tp, i.e., number of epochs that are **correctly** predicted as **anomalous**;
- True negatives tn, i.e, number of epochs that are correctly predicted as not anoma-

lous;

- False positives fp, i.e., number of epochs that are **incorrectly** predicted as **anomalous**; and
- False negatives fn, i.e., number of epochs that are **incorrectly** predicted as **not anomalous**.

The scenario ends by printing, in the standard output, the confusion matrix, the time needed to build the model and to make the prediction (print_metrics call).

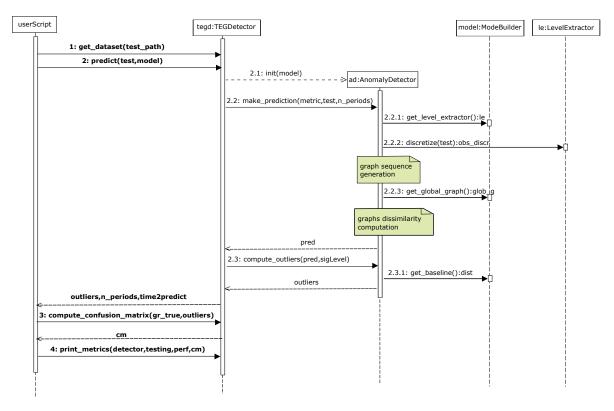


Figure 5: Anomaly detection and metric printing

4. Experiment: Smart grid fraud detection

Smart grids are complex cyber-physical systems built on top of electrical power infrastructures. Although resilient and reliable to supply electricity, smart grids are vulnerable to attacks and frauds (He and Yan 2016). Bernardi, Javierre, Merseguer, and Requeno (2021) studied different types of attacks on smart grids and evaluated the effectiveness of some anomaly detection techniques for these attacks. Now, we borrow some experiments in Bernardi et al. (2021) to illustrate how to build a prediction model and detect anomalies using the **tegdet** library.

4.1. Description of the dataset

The work of Bernardi *et al.* (2021) used datasets from the Ireland's Commission for Energy Regulation (ISSDA-CER). From them, synthetic datasets were created to model different types of attacks. Here, we use one of these synthetic datasets and some information related to a specific smart meter (ISSDA-CER).

Concretely, our dataset is made of real value observations, representing a smart meter energy consumption, in kWh, collected every half hour. The observation period is 75 weeks. The dataset is organized in three files³:

- training: Includes readings for the first 60 weeks. It will be used to build prediction models.
- test_normal: Includes readings for the remaining 15 weeks. Its observations are considered *normal*, i.e., it is assumed that they have not been affected by attacks. It will be used to detect anomalies.
- test_anomalous: This is a synthetic dataset created from the test_normal file, so it includes readings for the same period. It is assumed that its observations have been tampered to defraud the energy utility by paying less than consumed. It will also be used to detect anomalies.

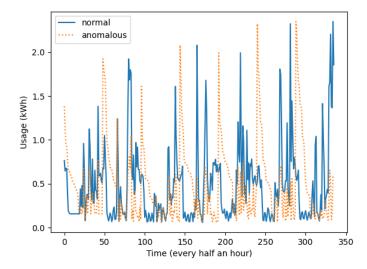
Figure 6 (left) represents the observations in the first week of test_normal and test_anomalous. The anomalous curve has been obtained by swapping the observations between peak and peak-off periods (from 9:00am to midnight and from midnight to 9:00am), so assuming to defraud a time-of-use contract having peak and off-peak periods (Krishna, Lee, Weaver, Iyer, and Sanders 2016). Figure 6 (right) offers an excerpt of the training file, which is in csv⁴ format. The first column indicates the time instant (observations are time ordered) and the second one the observed real value at that instant. The first row indicates the column headers. The tegdet library does not impose any restrictions on column names or time instant format.

4.2. How to build a prediction model and detect anomalies

The scenarios presented in Subsections 3.1 and 3.2 are now developed using Python code to illustrate how to carry out an experiment with the **tegdet** library. In particular, the Python

³The dataset is available in the GitHub repository: https://github.com/DiasporeUnizar/TEG

 $^{^4}$ Comma-separated value.



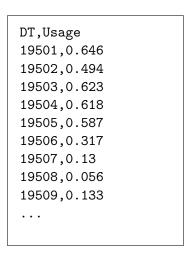


Figure 6: (left) Time series of the two testing sets, one week observations. (right) An excerpt of the training dataset

script in Appendix C.3, Listing 1, is used. Moreover, Appendix C.2 offers instructions for the installation of the required libraries to execute the script.

Initially, the script imports the API class TEGDetector from the tegdet library:

```
from tegdet.teg import TEGDetector
```

Then, the script sets the paths for the datasets and a results file:

```
TRAINING_DS_PATH = "/dataset/energy/training.csv "
TEST_DS_PATH = "/dataset/energy/test_"
RESULTS_PATH = "/script_results/energy/tegdet_variants_results.csv"
```

The script defines also the testing datasets and metrics that will be used:

The preliminaries of the script end by setting the input parameters, as indicated in Table 5 (Appendix C.1). Concretely, the energy consumption range is partitioned into 30 discretization levels, the epochs correspond to weeks (336 observations per period), and the predictions will be made considering a 95% significance level:

```
n_bins = 30
n_obs_per_period = 336
alpha = 5
```

Building the prediction model. The script builds a prediction model for a given metric in four steps: 1) it stores the path to the training dataset, 2) instantiates the TEGDetector class with the current metric, 3) loads the training dataset, and 4) builds the model:

```
def build_and_predict(metric):
    cwd = os.getcwd()
    train_path = cwd + TRAINING_DS_PATH
    tegd = TEGDetector(metric)
    train = tegd.get_dataset(train_path)
    model, time2build = tegd.build_model(train)
```

The output of the training dataset shows that the column headers have been renamed (TS, time-stamp and DP, data-point), and that it includes 20160 observations, during an overall period of 60 weeks:

```
>>> train
          TS
                 DP
0
       19501
              0.646
              0.494
1
       19502
2
       19503
              0.623
3
       19504
              0.618
4
       19505
              0.587
20155
       62144
              0.550
20156
       62145
              0.586
20157
       62146
              0.585
20158
       62147
              1.898
       62148 0.988
20159
[20160 rows x 2 columns]
```

Also as a result, the time to build the model (in seconds) is stored:

```
>>> time2build
0.1602308750152588
```

Anomaly detection. For each testing dataset, the script: 1) stores its path, 2) loads the dataset through the TEGDetector instance, and 3) carries out prediction for the current dataset and model:

```
for testing in list_of_testing:
          test_path = cwd + TEST_DS_PATH + testing + ".csv"
          test = tegd.get_dataset(test_path)
          outliers, n_periods, time2predict = tegd.predict(test, model)
```

As a result of the prediction, the script obtains: a) the outliers, an array where each entry corresponds to the prediction related to an epoch (i.e., 0 normal epoch, 1 anomalous epoch), b) the number of epochs (n_periods) and c) the time required to make the prediction (time2predict, in seconds).

The following snapshot presents the prediction of the Hamming detector when applied to the normal dataset. It detects one anomalous epoch in the overall set of 15 weeks, concretely the 4^{th} week (i.e., the 4^{th} entry in the array):

Next, the script collects results of the predictions. First, it computes the confusion matrix, using ground true values for each epoch, i.e., weeks. These values are obtained using the ones or zeros functions of the numpy library. Concretely, the normal dataset ground values are set to zero (i.e., its observations correspond to actual energy consumption), whereas the anomalous dataset ground values are set to one (i.e., its observations correspond to tampered consumptions):

The following snapshot presents the confusion matrix cm of the Hamming detector when applied to the normal testing dataset. A week is wrongly detected as anomalous (fp entry), while the rest of the weeks are correctly predicted as normal weeks (tn entry):

```
>>> cm
{'tp': 0, 'tn': 14, 'fp': 1, 'fn': 0}
```

Finally, the script: 1) prints, in the standard output, the confusion matrix and performance metrics, i.e., the time to build the model and the time to make prediction, and 2) stores the results in a **csv** file for post-processing purposes:

```
perf = {'tmc': time2build, 'tmp': time2predict}

tegd.print_metrics(detector, testing, perf, cm)

results_path = cwd + RESULTS_PATH

tegd.metrics_to_csv(detector, testing, perf, cm, results_path)
```

The following snapshot presents the results of the Hamming detector, considering both testing sets:

```
Detector:
                                      Hamming
N bins:
                                      30
N_obs_per_period:
                                      336
Alpha:
Testing set:
                                       normal
Time to build the model:
                                       0.1602308750152588 seconds
Time to make prediction:
                                       0.04300808906555176 seconds
Confusion matrix:
 {'tp': 0, 'tn': 14, 'fp': 1, 'fn': 0}
Detector:
                                       Hamming
```

```
N_bins: 30
N_obs_per_period: 336
Alpha: 5
Testing set: anomalous
Time to build the model: 0.1602308750152588 seconds
Time to make prediction: 0.059934139251708984 seconds
Confusion matrix:
{'tp': 10, 'tn': 0, 'fp': 0, 'fn': 5}
```

The following snapshot provides an excerpt from the results file when applying the Hamming detector to both testing datasets:

```
detector,n_bins,n_obs_per_period,alpha,testing_set,time2build,time2predict,tp,tn,fp,fn
Hamming,30,336,5,normal,0.1602308750152588,0.04300808906555176,0,14,1,0
Hamming,30,336,5,anomalous,0.1602308750152588,0.059934139251708984,10,0,0,5
```

5. Assessment of the library

The results file produced by the **tegdet** library can be post-processed to assess the quality of the very same library. In the following, using the experiment in previous section we assess: a) the performance of the algorithms implemented to build the prediction model and to predict the anomaly; and b) the *accuracy* of the anomaly detectors.

Concerning performance, Table 1 offers results regarding execution times⁵. On the one hand, we see that the mean time to build a model is approximately four times greater than the mean time to make a prediction. In fact, this is also the ratio between the lengths of the training dataset (60 weeks) and each testing dataset (15 weeks). On the other hand, the small standard deviations indicate that the times are not affected by the type of metric used to build the anomaly detector.

Statistical qualifier	Time to build (ms.)	Time to predict (ms.)
mean	174.46	47.70
std	11.17	3.58
\min	160.23	44.31
max	215.39	62.16

Table 1: Execution times statistics

Figure 7 shows the accuracy of the different anomaly detectors, which is defined as the ratio

 $^{^5\}mathrm{The}$ analysis has been carried out using a laptop with Intel Core i7 double core CPU, 16GB RAM, 250GB SSD.

of correct predictions to total predictions:

$$accuracy = \frac{(tp+tn)}{(tp+tn+fp+fn)} \tag{6}$$

For each anomaly detector, the accuracy has been computed considering the sum of two confusion matrices, obtained from the *normal* and *anomalous* testing sets, respectively. Observe that the resulting confusion matrix is balanced (i.e, tp + fp = fn + tn). Most of the detectors have high accuracy ($\geq 80\%$), being Clark and Divergence the bests (100% of correct predictions). Whereas, Dice (37%), Jaccard (37%) and Lorentzian (43%) exhibit the least accurate predictions for the considered datasets.

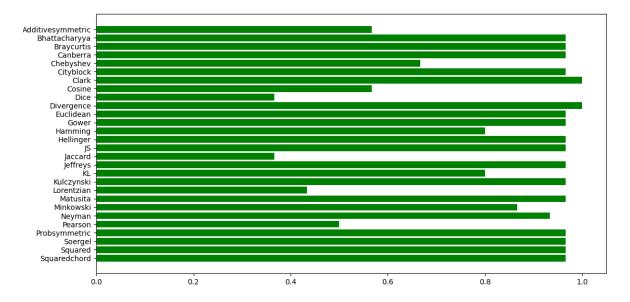


Figure 7: Accuracy of the anomaly detectors

6. Guidelines for improving predictions

An anomaly detector in **tegdeg** is characterized, besides the metric, by three parameters: 1) number of observations per period, 2) number of discretization levels and 3) α , significance level in the detection phase. An adequate setting of these parameters greatly impacts on the accuracy of the anomaly prediction, as well as on the time to build the model and to predict the anomaly. This section provides guidelines for such adequate setting of the parameters.

Table 2 shows the ranges considered from now on. The configuration used in Section 5 corresponds to the values in bold.

Performance. The times to build the model and predict the anomaly are mainly affected by the values of the n_obs_per_period and n_bins parameters. They define the characteristics of the graph sequences and the graph dissimilarity distribution.

Figure 8 offers 3D plots of the mean execution times of the Hamming detector considering these two parameters. The two surfaces have a similar shape, despite the time ranges, which

Parameter	Range
n_obs_per_period	[24, 48, 96, 168, 192, 336 , 480, 672, 816, 1008]
${\tt n_bins}$	[5, 10, 15, 20, 25, 30 , 35, 40, 45, 50]
alpha	[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Table 2: Parameters ranges

confirm the results in previous section, i.e., the times to build are approximately four time greater than the times to predict. The rest of the detectors exhibit similar results.

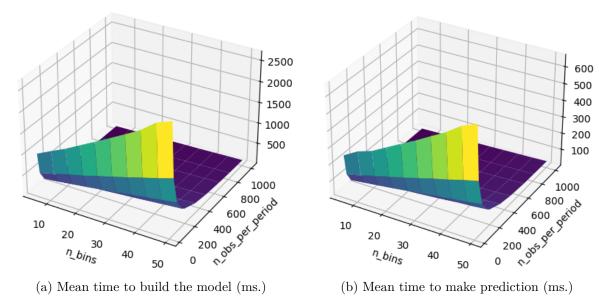


Figure 8: Performance results of the Hamming detector

We also observe that execution times are sensitive to both the parameters. In particular, the lower is the value of n_obs_per_period the higher is the execution time, since the longer will be the sequence of graphs to be generated. For example, changing the value of n_obs_per_period from 336 to 24 corresponds to change the epoch from 1 week to 12 hours, then the number of graphs to be generated for the training set will be 840, instead of 60 (for each testing set, 210 graphs instead of 15). Besides, the higher is the value of n_bins the higher is the execution time, since the size of the graphs to be generated (number of nodes and edges), as well as the computation of the graph dissimilarity, are in direct proportion to the number of discretization levels.

Prediction accuracy. The accuracy is computed as in Equation 6 applied to the *normal* and *anomalous* testing datasets.

Figure 9 shows 3D plots for the accuracy of two anomaly detectors, Hamming (plots on the left) and Clark (plots on the right). Each row considers the accuracy versus two of the three parameters. Then, fixing the third parameter to its default value (bold value in Table 2).

Overall, the experiments confirm the results in the Figure 7 for the default configuration, i.e.,

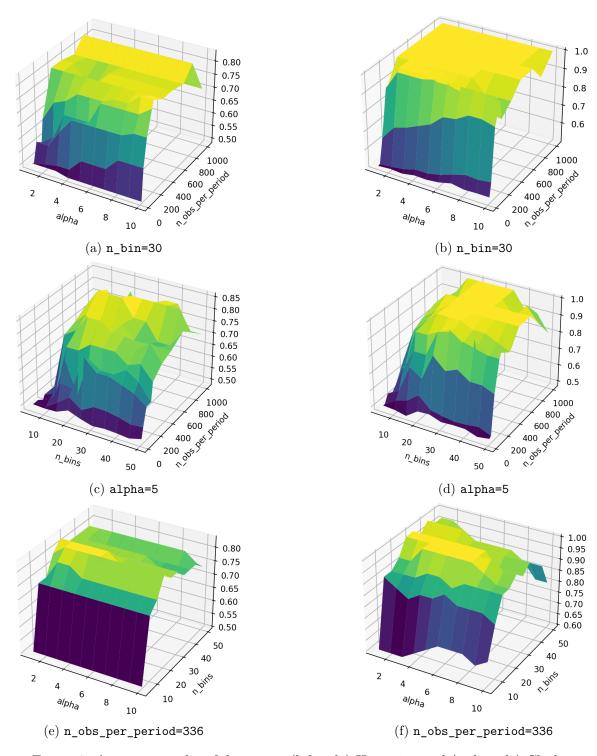


Figure 9: Accuracy results of detectors: (left side) Hamming and (right side) Clark

Clark outperforms Hamming. Indeed, the former is characterized by an accuracy range higher than the latter (cfr. the z-axis of the left plots vs/ the right ones). For some configurations, Clark accuracy reaches 100%, whereas the best value reached by Hamming is 85% (Figure 9c).

Moreover, Clark seems to be more stable than Hamming, since its surfaces are more flattened than the Hamming ones.

Let us consider the first row. Figures 9a and 9b show the accuracy versus the alpha and n_obs_per_period parameters. For both detectors, the accuracy seems to be slightly sensitive to the alpha parameter, which is used to compute the threshold for the anomaly detection phase), whereas it is highly sensitive to the n_obs_per_period. In particular, the lengths of the epochs between half-week (168 observations) and two weeks (672 observations) ensure better accuracy for both variants.

The second row shows the accuracy versus n_bin and n_obs_per_period parameters (Figures 9c and 9d). Both surfaces are characterized by a higher slope in the direction of n_obs_per_period, meaning that the influence of the second parameter is predominant regarding the first one. The accuracy is also sensitive to the n_bin parameter. In particular, values in the ranges [15, 30] and [20, 35] provide better accuracy for Hamming and Clark, respectively.

Finally, the third row shows the accuracy versus alpha and n_bin parameters. Concerning Hamming, Figure 9e confirms that the variability with respect to the alpha values is very low. Whereas, in the case of Clark (Figure 9f), the accuracy is slightly sensitive to this parameter and values between [3, 6] provide better results.

7. Conclusion

This first version of the **tegdet** library makes publicly available a Python API that enables to detect anomalies in univariate time series, by leveraging TEGs.

The **tegdet** library decouples the creation of the TEGs and the anomaly detection. This is important for two reasons. First, this is what promotes the extensibility of the library, allowing to implement new anomaly detection techniques, while the discretization and graph discovery steps remain the same. Appendix B explains how to accomplish the extensibility. Second, the decoupling also favours the good performance results of the library. Being the temporal complexity to create the TEGs linear, with respect to the length of the input, in the worst case, then, the performance impact of adding a new detection technique is attributable to the implementation of the technique exclusively.

However, this version of the library has also some limitations. The library currently detects anomalous epochs, i.e., graph-level anomalies. Next extensions will identify which parts of the anomalous graph contribute the most to the change, i.e., attribution. The input of the library is currently a univariate time series, hence multivariate time-series should be further investigated.

Going back to the library performance, the most time consuming activities belong to the anomaly detection: model construction and outlier detection. The worst case temporal complexity is O(m|E|), where m is the number of graphs and |E| the size of the global graph (number of edges). m depends on the length of the time-series N and the number of observations per epoch s (i.e., $m = \lfloor \frac{N}{s} \rfloor$), whereas |E| depends on the number of discretization levels L and can be bounded by $L-1 \leq |E| \leq L^2$. Therefore, the implemented anomaly detection technique is linear only with respect to the graph size (with a fixed m), as also proved by the results of the experiments — cfr. in Figure 8, the slope of the curves with respect to the axis n_bins (i.e., L) and n_obs_per_period (i.e., s). We are investigating whether graphs

sparse matrix implementations could even improve the (time and memory) efficiency of graph operations, such as the computation of graph dissimilarities.

The experiments presented here aim to provide guidelines for setting the parameters of the library. Therefore, the purpose is not to make a robust evaluation of the quality of the detectors for a specific application domain. Indeed, we have obtained the prediction accuracy results by considering a single time-series, i.e., the consumptions registered by one smart meter and a specific type of attack. Hence, although some detectors exhibit very good results, we cannot draw a general conclusion about their effectiveness. Furthermore, it is well-known that each application domain favours certain classes of detectors.

Finally, it is worth to say that our work is inspired by Ma et al. (2021), they propose a unified framework aimed at detecting different types of anomalies, for a specific application domain. In our view, the framework shall include datasets, synthetic dataset generators and implementations of detectors, and shall provide support for the comparative analysis of the quality of the detectors. The **tegdet** library could be part of such framework.

Acknowledgments

S. Bernardi and J. Merseguer were supported by the Spanish Ministry of Science and Innovation [ref. PID2020-113969RB-I00].

References

- Aggarwal CC, Zhao Y, Yu PS (2011). "Outlier detection in graph streams." In 2011 IEEE 27th International Conference on Data Engineering, pp. 399–409. doi:10.1109/ICDE. 2011.5767885.
- Akoglu L, Tong H, Koutra D (2015). "Graph Based Anomaly Detection and Description: A Survey." *Data Min. Knowl. Discov.*, **29**(3), 626–688. ISSN 1384-5810.
- Bernardi S, Javierre R, Merseguer J, Requeno JI (2021). "Detectors of Smart Grid Integrity Attacks: an Experimental Assessment." In 17th European Dependable Computing Conference, EDCC 2021, Munich Germany, September 13-16, 2021. IEEE Computer Society.
- Bhatia S, Hooi B, Yoon M, Shin K, Faloutsos C (2020). "Midas: Microcluster-Based Detector of Anomalies in Edge Streams." *Proceedings of the AAAI Conference on Artificial Intelligence*, **34**(04), 3242-3249. doi:10.1609/aaai.v34i04.5724. URL https://ojs.aaai.org/index.php/AAAI/article/view/5724.
- Cha SH (2007). "Comprehensive survey on distance/similarity measures between probability density functions." *International Journal of Mathematical Models and Methods in Applied Sciences*, **1**(4), 300–307.
- Chang YY, Li P, Sosic R, Afifi MH, Schweighauser M, Leskovec J (2021). "F-FADE: Frequency Factorization for Anomaly Detection in Edge Streams." In *Proceedings of the 14th ACM International Conference on Web Search and Data Mining*, WSDM '21, p. 589–597. Association for Computing Machinery, New York, NY, USA. ISBN 9781450382977. doi: 10.1145/3437963.3441806. URL https://doi.org/10.1145/3437963.3441806.

- Eswaran D, Faloutsos C (2018). "SedanSpot: Detecting Anomalies in Edge Streams." In *IEEE International Conference on Data Mining, ICDM 2018, Singapore, November 17-20, 2018*, pp. 953–958. IEEE Computer Society. doi:10.1109/ICDM.2018.00117. URL https://doi.org/10.1109/ICDM.2018.00117.
- Fan H, Zhang F, Li Z (2020). "AnomalyDAE: Dual autoencoder for anomaly detection on attributed networks." doi:10.48550/ARXIV.2002.03665. URL https://arxiv.org/abs/2002.03665.
- Febrinanto FG, Xia F, Moore K, Thapa C, Aggarwal C (2022). "Graph Lifelong Learning: A Survey." doi:10.48550/ARXIV.2202.10688. URL https://arxiv.org/abs/2202.10688.
- Gamma E, Helm R, Johnson R, Vlissides JM (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional. ISBN 0201633612.
- Grubbs FE (1969). "Procedures for Detecting Outlying Observations in Samples." *Technometrics*, **11**(1), 1–21. ISSN 00401706. URL http://www.jstor.org/stable/1266761.
- Gupta M, Gao J, Aggarwal CC, Han J (2014). Outlier Detection for Temporal Data. Synthesis Lectures on Data Mining and Knowledge Discovery. Morgan & Claypool Publishers. doi:10.2200/S00573ED1V01Y201403DMK008. URL https://doi.org/10.2200/S00573ED1V01Y201403DMK008.
- Hamming RW (1950). "Error detecting and error correcting codes." The Bell System Technical Journal, 29(2), 147–160. doi:10.1002/j.1538-7305.1950.tb00463.x.
- Hawkins D (1980). *Identification of outliers*. Monographs on applied probability and statistics. Chapman and Hall, London [u.a.]. ISBN 041221900X.
- He H, Yan J (2016). "Cyber-physical attacks and defences in the smart grid: a survey." *IET Cyber-Physical Systems: Theory & Applications*, **1**(1), 13–27.
- Hooi B, Akoglu L, Eswaran D, Pandey A, Jereminov M, Pileggi L, Faloutsos C (2018). "ChangeDAR: Online Localized Change Detection for Sensor Data on a Graph." In Proceedings of the 27th ACM International Conference on Information and Knowledge Management, CIKM '18, p. 507–516. Association for Computing Machinery, New York, NY, USA. ISBN 9781450360142. doi:10.1145/3269206.3271669. URL https://doi.org/10.1145/3269206.3271669.
- ISSDA-CER (2012).CERSmartMetering Project. Social Sci-Commission URL: ence Data Archive, for Energy Regulation. https://www.ucd.ie/issda/data/commissionforenergyregulationcer/.
- Jeffreys H (1961). Theory of Probability. Third edition. Oxford, Oxford, England.
- Krishna VB, Lee K, Weaver GA, Iyer RK, Sanders WH (2016). "F-DETA: A Framework for Detecting Electricity Theft Attacks in Smart Grids." In 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 407–418.
- Li J, Han Z, Cheng H, Su J, Wang P, Zhang J, Pan L (2019). "Predicting Path Failure In Time-Evolving Graphs." In *Proceedings of the 25th ACM SIGKDD International Conference on*

- Knowledge Discovery & Data Mining, KDD '19, p. 1279–1289. Association for Computing Machinery, New York, NY, USA. ISBN 9781450362016. doi:10.1145/3292500.3330847. URL https://doi.org/10.1145/3292500.3330847.
- Liu Y, Pan S, Wang YG, Xiong F, Wang L, Chen Q, Lee VC (2021). "Anomaly Detection in Dynamic Graphs via Transformer." *IEEE Transactions on Knowledge and Data Engineering*, pp. 1–1. doi:10.1109/TKDE.2021.3124061.
- Ma X, Wu J, Xue S, Yang J, Zhou C, Sheng QZ, Xiong H, Akoglu L (2021). "A Comprehensive Survey on Graph Anomaly Detection with Deep Learning." *IEEE Transactions on Knowledge and Data Engineering*, pp. 1–1. doi:10.1109/TKDE.2021.3118815.
- Shumway RH, Stoffer DS (2011). Time Series Analysis and Its Applications With R Examples. Springer Texts in Statistics, 3 edition. Springer Science+Business Media.
- Stroustrup B (2013). The C++ programming language. Pearson Education.
- Teng X, Yan M, Ertugrul AM, Lin YR (2018). "Deep into Hypersphere: Robust and Unsupervised Anomaly Discovery in Dynamic Networks." In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pp. 2724–2730. International Joint Conferences on Artificial Intelligence Organization. doi:10.24963/ijcai. 2018/378. URL https://doi.org/10.24963/ijcai.2018/378.
- van Rossum G (2007). "Python Programming Language." In J Chase, S Seshan (eds.), Proceedings of the 2007 USENIX Annual Technical Conference, Santa Clara, CA, USA, June 17-22, 2007. USENIX.
- Wu J, Pan S, Zhu X, Zhang C, Yu PS (2018). "Multiple Structure-View Learning for Graph Classification." *IEEE Transactions on Neural Networks and Learning Systems*, **29**(7), 3236–3251. doi:10.1109/TNNLS.2017.2703832.
- Yoon M, Hooi B, Shin K, Faloutsos C (2020). "Fast and Accurate Anomaly Detection in Dynamic Graphs with a Two-Pronged Approach." doi:10.48550/ARXIV.2011.13085. URL https://arxiv.org/abs/2011.13085.
- Zheng L, Li Z, Li J, Li Z, Gao J (2019). "Addgraph: Anomaly Detection in Dynamic Graph Using Attention-Based Temporal GCN." In Proceedings of the 28th International Joint Conference on Artificial Intelligence, IJCAI'19, p. 4419–4425. AAAI Press. ISBN 9780999241141.
- Zong B, Song Q, Min MR, Cheng W, Lumezanu C, Cho D, Chen H (2018). "Deep Autoencoding Gaussian Mixture Model for Unsupervised Anomaly Detection." In *International Conference on Learning Representations*. URL https://openreview.net/forum?id=BJJLHbb0-.

A. Dissimilarity metrics

Table 3 lists the metrics currently implemented in the **tegdet** library, which are used to compute dissimilarities between two graphs. The table groups the metrics by families, as given by Cha (2007). All the metrics are defined based on two n-length vectors, say P and Q. When applied to graph dissimilarity computation, the Hamming metric considers only the structural characteristics of the graphs, then P and Q are obtained by just flattening the adjacency matrix of each graph. In the case of the Cosine metric, P and Q are obtained by flattening the node frequency vector and the adjacency matrix of each graph. Finally, for the rest of the metrics, P and Q are obtained by flattening the adjacency matrix of each graph, where the entries represent absolute frequencies, and converting the resulting vectors into relative frequencies.

Metric	Definition	
Graph edit distance family		
Hamming	$1 - \frac{\sum_{i=1}^{n} \chi(P_i, Q_i)}{n}$	
	where $\chi(x,y) = \begin{cases} 1 & \text{if } (x > 0 \land y > 0) \lor (x = 0 \land y = 0) \\ 0 & \text{otherwise} \end{cases}$	
Inner product family		
Cosine	$1 - \frac{\sum_{i=1}^{n} P_i \cdot Q_i}{\sqrt{\sum_{i=1}^{n} P_i^2} \cdot \sqrt{\sum_{i=1}^{n} Q_i^2}}$	
Jaccard	$\frac{\sum_{i=1}^{n} (P_i - Q_i)^2}{(\sum_{i=1}^{n} P_i^2 + \sum_{i=1}^{n} Q_i^2 - \sum_{i=1}^{n} P_i \cdot Q_i)}$ $\frac{\sum_{i=1}^{n} (P_i - Q_i)^2}{(\sum_{i=1}^{n} P_i^2 + \sum_{i=1}^{n} Q_i^2)}$	
Dice	$\frac{\sum_{i=1}^{n} (P_i - Q_i)^2}{(\sum_{i=1}^{n} P_i^2 + \sum_{i=1}^{n} Q_i^2)}$	
Shannon's entropy family		
KL	$\sum_{i=1}^{n} P_i \cdot log_2 \frac{P_i}{Q_i}$	
Jeffreys	$\sum_{i=1}^{n} (P_i - Q_i) \cdot ln \frac{P_i}{Q_i}$	
JS	$\sqrt{\frac{KL(P,M)+KL(Q,M)}{2}}$ where $KL(\cdot,\cdot)$ is the KL function and M is the pointwise mean vector of P and Q	
	L_p Minkowski family	
Euclidean	$\sqrt{\sum_{i=1}^{n} P_i-Q_i ^2}$	
Cityblock	$\sum_{i=1}^{n} P_i - Q_i $	
Chebyshev	$max_{i=1}^{n} P_i - Q_i $	
Minkowski (p=3)	$\sqrt[p]{\sum_{i=1}^{n} P_i-Q_i ^p}$	
L_1 family		
Braycurtis (Sørensen)	$\frac{\sum_{i=1}^{n} P_i - Q_i }{\sum_{i=1}^{n} (P_i + Q_i)}$	
Gower	$\frac{1}{n}\sum_{i=1}^{n} P_i-Q_i $	

Soergel	$\frac{\sum_{i=1}^{n} P_i - Q_i }{\sum_{i=1}^{n}max(P_i,Q_i)}$	
Kulczynski	$\frac{\sum_{i=1}^{n} P_i-Q_i }{\sum_{i=1}^{n}min(P_i,Q_i)}$	
Canberra	$\sum_{i=1}^{n} \frac{ P_i - Q_i }{(P_i + Q_i)}$	
Lorentzian	$\sum_{i=1}^{n} ln \ (1 + P_i - Q_i)$	
Fidelity or Squared-chord family		
Bhattacharyya	$-ln \sum_{i=1}^{n} \sqrt{P_i \cdot Q_i}$	
Hellinger	$2\sqrt{1-\sum_{i=1}^{n}\sqrt{P_i\cdot Q_i}}$	
Matusita	$\sqrt{2-2\sum_{i=1}^{n}\sqrt{P_i\cdot Q_i}}$	
Squaredchord	$\sum_{i=1}^{n} (\sqrt{P_i} - \sqrt{Q_i})^2$	
Squared L_2 or χ^2 family		
Pearson χ^2	$\sum_{i=1}^{n} \frac{(P_i - Q_i)^2}{Q_i}$	
Neyman χ^2	$\sum_{i=1}^{n} \frac{(P_i - Q_i)^2}{P_i}$	
Squared χ^2	$\sum_{i=1}^{n} \frac{(P_i - Q_i)^2}{P_i + Q_i}$	
Prob. Symmetric χ^2	$2\sum_{i=1}^{n} \frac{(P_i - Q_i)^2}{P_i + Q_i}$	
Divergence	$2\sum_{i=1}^{n} \frac{(P_i - Q_i)^2}{(P_i + Q_i)^2}$	
Clark	$\sqrt{\sum_{i=1}^{n} \left(\frac{ P_i - Q_i }{P_i + Q_i}\right)^2}$	
Additive Symmetric χ^2	$\sum_{i=1}^{n} \frac{(P_i - Q_i)^2 \cdot (P_i + Q_i)}{P_i \cdot Q_i}$	

Table 3: Dissimilarity metrics implemented in the **tegdet** library

B. Extensibility of the library

The **tegdet** library has been designed for being extensible. In particular, the library can be extended to support new classes of dissimilarity metrics, in addition to the 28 already implemented. The importance of implementing new metrics strives on augmenting the capabilities of the library to introduce new anomaly detector techniques. The extensibility has been achieved by applying a variant of the strategy design pattern (Gamma, Helm, Johnson, and Vlissides 1994) to the software design of the library, see Figure 3 (Section 3).

Since dissimilarity metrics define techniques used to compare graphs, then an abstract class, named GraphComparator, is proposed that will be specialized in as many classes as techniques want to be implemented. Figure 3 depicts three specialized classes as examples.

An abstract class owns one or more abstract methods, in this case compare_graphs is the abstract method that needs to be implemented in each of the specialized classes, having the purpose of implementing the technique defined by the dissimilarity metric. Hence, a user of

the library wanting to extend it with a new metric, say <myMetric>, only needs to create a new sub-class of GraphComparator, name it as Graph<myMetric>Dissimilarity, and implement the technique in the compare_graphs method.

Finally, when a user wants to generate a TEG detector based on the new metric, she/he creates a new instance of the TEGDetector class by setting the metric parameter to the name of the new metric in the __init()__ instantiation method — see Figure 4, message 1 (Section 3).

C. Library repositories

The **tegdet** library is available at the official PyPi repository⁶ as well as at the GitHub repository⁷. In particular, the latter also includes the following resources:

- API documentation,
- Documentation about the installation and implementation,
- Dataset used in Section 4,
- Test and example Python scripts.

C.1. API of the library

Table 4 defines the API of the library, while Table 5 details the attributes of the TEGDetector class, which are the parameters of the constructor of the API.

C.2. Library installation

The **tegdet** library has been implemented in Python3 and it runs with Python versions $\geq 3.6.1$. The library can be easily installed from the PyPi repository as follows:

> pip3 install tegdet

As an alternative, the *GitHub* repository can be either cloned (or downloaded) and the library can be installed from the distribution local to such repository using the command:

```
> pip3 install dist/tegdet-1.0.0-py3-none-any.whl
```

Since the library depends on the **pandas**, **numpy** and **scipy** Python packages, these should be installed before using the library.

The library dependencies are also listed in the requirements.txt file (available in the *GitHub* repository), and all the necessary packages can be installed using the command:

> pip3 install -r requirements.txt

⁶PyPi url: https://pypi.org/project/tegdet/

⁷GitHub url: https://github.com/DiasporeUnizar/TEG

tegdet API methods	Description
<pre>init(metric: string, n_bins: int =_N_BINS, n_obs_per_period: int =_N_OBS_PER_PERIOD, alpha: int =_ALPHA)</pre>	Constructor that initializes the TEGDetector input parameters (see Table 5).
<pre>get_dataset(ds_path: string): DataFrame</pre>	Loads the dataset from ds_path file (comma-separated values format), renames the columns and returns it as a pandas Dataframe.
<pre>build_model(training_dataset: Dataframe): ModelBuilder, float</pre>	Builds the prediction model based on the training_dataset and returns it (ModelBuilder object) together with the time to build the model (float type).
<pre>predict(testing_dataset: Dataframe, model: ModelBuilder): numpy array of int, int, float</pre>	Makes predictions on the testing_dataset using the model. It returns: the outliers (numpy array of $\{0,1\}$ values) and total number of observations (int type), and the time to make predictions (float type).
<pre>compute_confusion_matrix(ground_true: numpy array of int, predictions: numpy array of int): dict</pre>	Computes the confusion matrix based on the ground true values and predicted values (numpy array of {0,1} values). It returns the confusion matrix as a dictionary (dict) type.
<pre>print_metrics(detector: dict, testing_set: string, perf: dict, cm: dict)</pre>	Prints on the standard output: the detector (dict type including the metric and the input parameter setting), the testing_set, the performance metrics perf (dict type including the time to build the model and the time to make predictions) and the confusion matrix cm.
<pre>metrics_to_csv(detector: dict, testing_set: string, perf: dict, cm: dict, results_csv_path: string)</pre>	Saves in the file with pathname results_csv_path (comma- separated values format): the detector (dict type), the testing_set, the performance metrics perf (dict type) and the confusion matrix cm (dict type).

Table 4: tegdet API (TEGDetector methods)

Input parameters	Description
metric: string n_bins: int n_obs_per_period: int alpha: int	Dissimilarity metric used to compare two graphs. Level of discretization (number of levels). Default value=N_BINS (=30) Number of observation per period. Default value=N_OBS_PER_PERIOD (=336) Significance level: 100-alpha. Default value=ALPHA (=5)

Table 5: TEGDetector attributes

C.3. Example scripts

The *GitHub* repository includes all the scripts and the dataset, which are necessary to reproduce the analysis carried out in Sections 4 and 6. In particular, the examples/energy folder includes the following scripts:

- tegdet_variants.py: builds and make predictions with TEG-detector variants (Subsection 4.2). The code is shown in Listing 1;
- tegdet_params_sensitivity.py: carries out sensitivity analysis of TEG-detector parameters (Section 6);
- post_processing.py: produces reports from the dataset (Sub-section 4.1) and the results of the analysis (Sections 5 and 6).

The first two scripts are examples of using the API, they both use the files in dataset/energy folder, and produce a results file <name_of_the_script>_results.csv in the script_results/

energy folder.

The script post_processing.py should be executed after the other two scripts. Indeed, it relies on both, the files in the dataset/energy folder and in the script_results/energy folder to produce reports (comparison of the testing datasets, performance and accuracy of the TEG-detectors). Since the script generates 3D plots, it is necessary to install the following package before running it:

> pip3 install matplotlib

All the scripts can be run using the following command from the root directory of the repository⁸:

```
> python3 examples/energy/<name_of_the_script>.py
```

Reproducibility of the results. The execution time results presented in Sections 4, 5 and 6 (i.e., time to build a model and time to make prediction) depend on the execution environment. Therefore, to reproduce such results, the script post_processing.py includes the following lines:

```
TEGDET_VARIANTS_RESULTS_PATH = "/script_results/energy/tegdet_variants_results_ref.csv"
TEGDET_PARAMS_SENSITIVITY_RESULTS_PATH = "/script_results/energy/tegdet_params_sensitivity_results_ref.csv"
#Uncomment these lines to analyse the results once the scripts have been run
#TEGDET_VARIANTS_RESULTS_PATH = "/script_results/energy/tegdet_variants_results.csv"
#TEGDET_PARAMS_SENSITIVITY_RESULTS_PATH = "/script_results/energy/tegdet_variants_results.csv"
```

The first two lines set the result paths to the files generated during the analysis described in Sections 4, 5 and 6, by the scripts tegdet_variants.py and tegdet_params_sensitivity.py, respectively. The last two lines set the result paths to the files that will be generated by a new execution of the above scripts.

Listing 1: Script tegdet_variants.py

```
1
    import os
    import pandas as pd
    import numpy as np
    from tegdet.teg import TEGDetector
 4
6
    #Input datasets/output results paths
    TRAINING_DS_PATH = "/dataset/energy/training.csv"
 7
 8
    TEST_DS_PATH = "/dataset/energy/test_"
    RESULTS_PATH = "/script_results/energy/tegdet_variants_results.csv"
9
10
11
     #List of testing
    list_of_testing = ("normal", "anomalous")
12
13
14
    #List of metrics (detector variants)
    list_of_metrics = ("Hamming", "Cosine", "Jaccard", "Dice", "KL", "Jeffreys", "JS",
15
                              "Euclidean", "Cityblock", "Chebyshev", "Minkowski", "Braycurtis",
16
                              "Gower", "Soergel", "Kulczynski", "Canberra", "Lorentzian",
"Bhattacharyya", "Hellinger", "Matusita", "Squaredchord",
"Pearson", "Neyman", "Squared", "Probsymmetric", "Divergence",
"Clark", "Additivesymmetric")
17
18
19
20
21
    #Parameters: default values
23
    n bins = 30
```

⁸The PYTHONPATH environment variable has to be set to the root directory of the tegdet project before running the scripts.

```
n_obs_per_period = 336
   alpha = 5
26
27
   def build_and_predict(metric):
28
       cwd = os.getcwd()
       train_path = cwd + TRAINING_DS_PATH
29
30
31
       tegd = TEGDetector(metric)
32
       #Load training dataset
33
       train = tegd.get_dataset(train_path)
34
       #Build model
35
       model, time2build = tegd.build_model(train)
36
37
       for testing in list_of_testing:
            #Path of the testing
39
            test_path = cwd + TEST_DS_PATH + testing + ".csv"
40
41
            #Load testing dataset
            test = tegd.get_dataset(test_path)
42
43
            #Make prediction
           outliers, n_periods, time2predict = tegd.predict(test, model)
44
45
            #Set ground true values
46
            if testing == "anomalous":
                ground_true = np.ones(n_periods)
47
48
            else:
49
                ground_true = np.zeros(n_periods)
50
           #Compute confusion matrix
52
            cm = tegd.compute_confusion_matrix(ground_true, outliers)
53
            #Collect detector configuration
            detector = {'metric': metric, 'n_bins': n_bins,
55
                            'n_obs_per_period':n_obs_per_period, 'alpha': alpha}
56
57
            #Collect performance metrics in a dictionary
            perf = {'tmc': time2build, 'tmp': time2predict}
58
59
60
            #Print and store basic metrics
61
            tegd.print_metrics(detector, testing, perf, cm)
62
            results_path = cwd + RESULTS_PATH
            tegd.metrics_to_csv(detector, testing, perf, cm, results_path)
63
64
65
   if __name__ == '__main__':
66
67
        for metric in list_of_metrics:
            build_and_predict(metric)
68
```