Anomaly Detection in Time Series Data: Unsupervised Learning, Isolation Forest

By

Pankaj Solanki (Data Analytics Intern)

From

DayalBagh Educational Institute

Submitted to

Supervisor: Mr. Guru Charan Bulusu (SWE)

From

Microsoft

1. INTRODUCTION

The scope of this project is to conduct a comprehensive analysis of anomaly detection techniques on time series data, with a particular focus on the *'machine_temperature_system_failure'* time series from the *realKnownCause* dataset.

1.1 The objectives of the project are as follows:

- **Data Pre-processing:** Implement data pre-processing techniques to clean and prepare the time series data for analysis. Handle missing data, normalize the values, and address any other data quality issues

- **Exploratory Data Analysis (EDA):** Perform exploratory data analysis on the 'machine_temperature_system_failure' time series to gain insights into its characteristics and patterns. Visualize the data to identify any noticeable trends or anomalies.

- **Anomaly Detection Modelling:** Utilize various anomaly detection algorithms, including Isolation Forest, LOF, SVM, and Microsoft Anomaly Detection service, to identify anomalies in the time series data. Implement and compare these algorithms using the F1 score as the evaluation metric.

- **Benchmark Evaluation:** Evaluate the performance of the anomaly detection algorithms on the NAB-dataset, which serves as a benchmark for assessing anomaly detection algorithms in different fields. Analyse the results and determine the strengths and weaknesses of each technique.

- **Business Application:** Apply the insights gained from the anomaly detection analysis to address specific business interests related to the 'machine_temperature_system_failure' time series. Discuss potential applications and actionable recommendations based on the anomaly detection results.

1.2 Dataset Selection:

- Choose the 'machine_temperature_system_failure' time series from the realKnownCause dataset, as it aligns with the project's business interests and provides a realistic scenario for anomaly detection.
- Utilize the NAB-dataset, comprising 58 time series data from various sources, to evaluate the performance of the anomaly detection algorithms comprehensively.

1.3 Steps Performed:

- Perform data pre-processing, including handling missing values and normalization, to ensure the data is ready for analysis.
- Conduct exploratory data analysis to gain insights into the 'machine_temperature_system_failure' time series and visualize any patterns or anomalies.

- Implement and compare Isolation Forest, LOF, SVM, and Microsoft Anomaly Detection service to detect anomalies, using the F1 score as the primary evaluation metric.
- Utilize visualization techniques to present the results effectively and enhance the understanding of the detected anomalies.

Overall, this project aims to showcase the process of anomaly detection on time series data, from data pre-processing and exploratory analysis to the evaluation of multiple algorithms, providing valuable insights for practical applications in real-world scenarios.

1.4 Related Terms

An Anomaly is a data point that is significantly different from the remaining data. Hawkins defined an Anomaly as follows:

"An Anomaly is an observation which deviates so much from the other observations as to arouse suspicions that it was generated by a different mechanism."

Anomalies are also referred to as abnormalities, discordant, deviants, or outliers in the data mining and statistics literature. In most applications, the data is created by one or more generating processes, which could either reflect activity in the system or observations collected about entities. When the generating process behaves unusually, it results in the creation of Anomaly. Therefore, an Anomaly often contains useful information about abnormal characteristics of the systems and entities that impact the data generation process. The recognition of such unusual characteristics provides useful application-specific insights.

Some examples are as follows:

- **Intrusion detection systems:** In many computer systems, different types of data are collected about the operating system calls, network traffic, or other user actions. This data may show unusual behaviour because of malicious activity. The recognition of such activity is referred to as intrusion detection.
- **Credit-card fraud:** Credit-card fraud has become increasingly prevalent because of greater ease with which sensitive information such as a credit-card number can be compromised. In many cases, unauthorized use of a credit card may show different patterns, such as buying sprees from particular locations or very large transactions. Such patterns can be used to detect outliers in credit-card transaction data**.**
- **Interesting sensor events:** Sensors are often used to track various environmental and location parameters in many real-world applications. Sudden changes in the underlying patterns may represent events of interest. Event detection is one of the primary motivating applications in the field of sensor networks. As discussed later in this book, event detection is an important temporal version of outlier detection.
- **Medical diagnosis:** In many medical applications, the data is collected from a variety of devices such as magnetic resonance imaging (MRI) scans, positron emission tomography (PET) scans or electrocardiogram (ECG) time-series. Unusual patterns in such data typically reflect disease conditions.

In all these applications, the data has a "normal" model, and anomalies are recognized as deviations from this normal model. Normal data points are sometimes also referred to as inliers. In some applications such as intrusion or fraud detection, outliers correspond to sequences of multiple data points rather than individual data points. For example, a fraud event may often reflect the actions of an individual in a particular sequence. The specificity of the sequence is relevant to identifying the anomalous event. Such anomalies are also referred to as collective anomalies, because they can only be inferred collectively from a set or sequence of data points. Such collective anomalies are often a result of unusual events that generate anomalous patterns of activity. This book will address these different types of anomalies.

The output of an outlier detection algorithm can be one of two types:

- **Outlier scores:** Most Anomaly detection algorithms output a score quantifying the level of "outlierness" of each data point. This score can also be used to rank the data points in order of their outlier tendency. This is a very general form of output, which retains all the information provided by a particular algorithm, but it does not provide a concise summary of the small number of data points that should be considered outliers.

- **Binary labels:** A second type of output is a binary label indicating whether a data point is an outlier or not. Although some algorithms might directly return binary labels, outlier scores can also be converted into binary labels. This is typically achieved by imposing thresholds on outlier scores, and the threshold is chosen based on the statistical distribution of the scores. A binary labelling contains less information than a scoring mechanism, but it is the final result that is often needed for decision making in practical applications.

How Anomalies are Categorized in Time Series Data: -

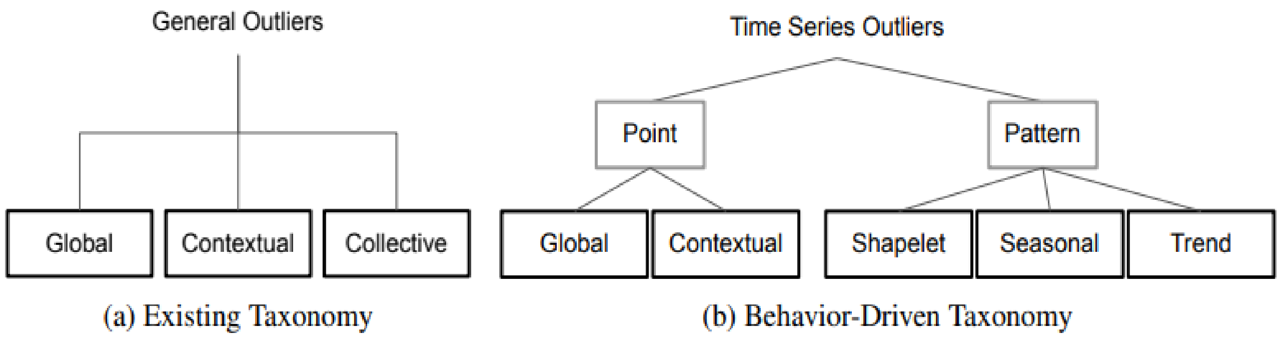


Figure 1*: Comparison of the behaviour-driven taxonomy with the existing taxonomy. We categorize sequential outliers into point and patten-wise behaviours with clear definitions of contexts.*

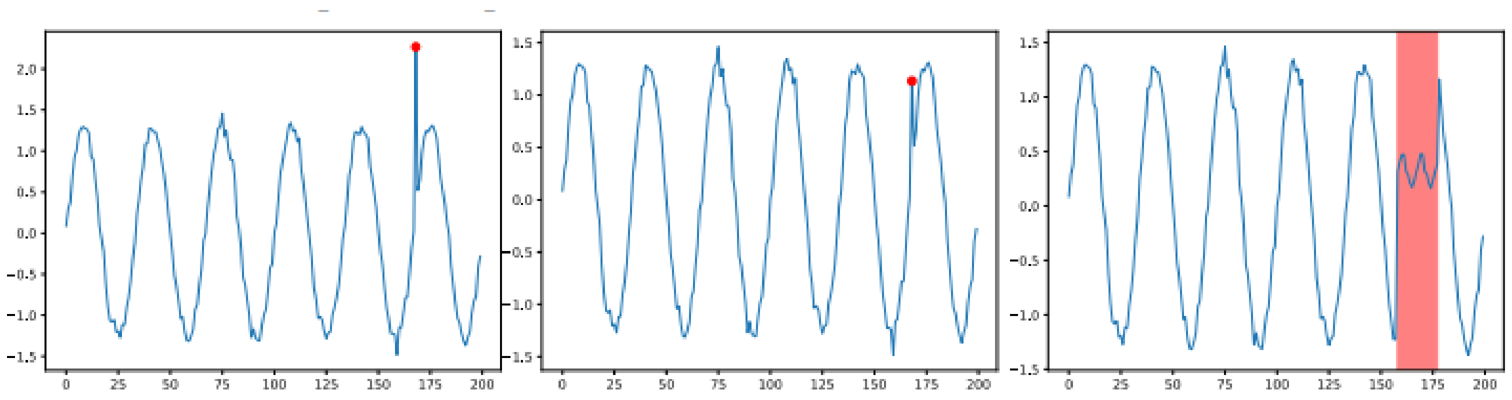


***Figure 2**: Examples of point (left), contextual (middle), and collective (right) outliers.*

2. Background

This section gives a background of the previous outlier definitions in time series data. Outliers in non-sequential data are often defined as the data instances that significantly deviate from the majority of the instances. However, it is non-trivial to define outliers in time series data due to the temporal correlations among observations. Existing studies often follow the outlier definitions in non-sequential data. Specifically, they define the outliers in sequential data with behaviour analysis and categorize them into point, contextual, and collective outliers. Figure 3 illustrates the three types of outliers that often serve as a de-facto-standard:

- **Point outlier** is defined as the individual instance that is anomalous with respect to the rest of the data. The extreme values could lead to serious consequences, and therefore point outlier is often the focus of sequential outlier detection research.

- **Contextual outlier** is the individual instance that is anomalous under a specific context, such as the discord points within the same harmonic pattern. Contextual outliers usually have relatively larger/smaller values in their own context but not globally. Identifying contextual outliers is often considered more challenging and is extensively explored in the literature.

- **Collective outlier** is defined as a collection of related data instances that is anomalous with respect to the entire data set. Specifically, the individual points of a collective outlier may not be anomalous by themselves but the co-occurrence of them becomes an outlier. Collective outliers are ubiquitous in sequential data since there are often strong dependencies among time points.

Although the above categorization has covered both individual instances and collections of instances, it remains non-trivial to clearly define the collective and contextual outliers due to the ambiguity of contexts. The contexts of the contextual outliers are often very different in the literature. They can be a small window containing the neighbouring points or the points with similar relative positions in terms of seasonality. Similarly, collective outliers can only be clearly defined with a clear context. For example, in Figure 1, the shapelet, seasonal and trend outliers have totally different behaviours under different contexts. However, the current taxonomy will categorize all of them as collective outliers since they are all outliers for multiple time points. To bridge this gap, this work aims to refine the sequential outlier definitions with clear and unified definitions of the contexts.

3. Revisiting Outlier Definition and Synthesizing Criteria

This section introduces a new taxonomy for time series outliers. We first revisit and motivate the behaviours of time series data with empirical observations and spectral analysis. Then we propose a new taxonomy for point- and pattern-wise outliers with clear context definitions. Finally, we discuss the existing synthetic methods and present a general synthetic criterion based on our new definitions.

3.1 Behaviours in Sequential Data

The most common way to model time series data is based on empirical observation, which treats the data as a series of data points and studies the relationships among the points. Formally, a time series data X with t timestamps can be represented as an ordered sequence of data points:

$$X = (x1, x2, \cdots, xt) \tag{1}$$

where x_i is the data point at timestamp i ($i \in T$, where $T = \{1, 2, \ldots, t\}$). This formulation can be naturally extended to a multivariate counterpart by adding a dimension to x_i. Previous work often follows empirical observation to study point, contextual, or collective outliers However, such formulation does not consider the temporal structure of the data such as trend and seasonal information. For example, given two anomalous sub sequences with unusual shapelet and abnormally high frequency respectively, they will be both identified as collective outliers. This makes it difficult to analyse the cause of outliers and understand the performance of detection algorithms.

To better model the temporal structure of the time series data, we can alternatively view the data with spectral analysis. The most common way of spectral analysis is to formulate the time series data as a combination of sinusoidal wave $X = \sum n \, Asin(2\pi\omega nT) + Bcos(2\pi\omega nT)$, were $sin(2\pi\omega nT)$ and $cos(2\pi\omega nT)$ are shapelet functions that transform a series of timestamps $T = \{1, 2, \ldots, t\}$ into values, and A and B are coefficients to define the value range. X is obtained by summing up the values of multiple waves with different frequencies, and ωn denotes the frequency of wave n. Although the sinusoidal wave can well represent the shapelets and seasonality of the data, it cannot model trend. To tackle this issue, we adopt structural modelling with spectral analysis to represent the time *series* as the combination of trend, seasonality and shapelets:

$$X = \rho(2\pi\omega T) + \tau(T) \tag{2}$$

Where $\rho(2\pi\omega T) = \sum n \, Asin(2\pi\omega nT) + Bcos(2\pi\omega nT)$] is the base shapelet function to approximate de-trend series here $\omega = \{\omega 1, \omega 2, \ldots., \omega n\}$ for brevity, and $\tau(.)$ denotes a trend function that models the general direction of the series. This formulation can represent various shapelet patterns, such as sawtooth wave and square wave, with various trends. For example, for square sine wave with linearly increasing trend, we can set $A = \frac{1}{2n+1}$, $B = 0$, $\omega n = 2n + 1$ ($n \in \{0,1, \dots, N\}$) , and τ as a linear function $\tau(T) = T$, where N controls the level of squareness.

3.2 Refining Sequential Anomalies Definitions

The existing taxonomy for time series data mainly focuses on individual data points, e.g., point and contextual outliers. While collective outlier considers subsequence, it simply regards a subsequence as a combinatorial behaviour of multiple points, which ignores the spectral information of subsequence. In this subsection, we propose a new taxonomy, shown in Figure 1b. We refine the outlier definitions in time series and identify five types of outliers that cover point- and pattern-wise behaviours.

3.2.1 Point-wise Anomalies

Point-wise Anomalies refer to unexpected incidents on individual time points. Anomalous behaviours of one time point can be a glitch or spike, where spike is an individual point with extreme value comparing to the rest of the points and glitch is an individual point with relatively deviated value from its neighbouring points. Following this intuition, given a time series $X = (x1, x2, \dots\dots, xt)$, two outlier types can be defined under point-wise behaviour with different thresholds δ :

$$| xt - \hat{x}t | > \delta, \tag{3}$$

Where $\hat{x}t$ is the expected value, which can be the output of a regression model, or simply the global mean value or mean value of a context window.

Global Anomalies refers to the points that significantly deviate from the rest of the points. They are usually the spikes in the series and therefore the threshold can be defined as

$$\delta = \lambda . \sigma(X) \tag{4}$$

Where $\sigma(X)$ is the standard deviation of the time series and λ controls the range.

Contextual Anomalies are the points that deviate from its corresponding context, which is defined as the neighbouring time points within certain ranges. This type of outlier are the small glitches in the sequential data and can be defined as:

$$\delta = \lambda . \sigma(X_{t-k,t+k}) \tag{5}$$

Where $X_{t-k,t+k} = (x_{t-k,} x_{t-k+1,} x_{t-k+2} \ . \ . \ .. \ x_{t+k})$ refers to the context of the data point $x_t$with a context window size k, and λ controls the threshold.

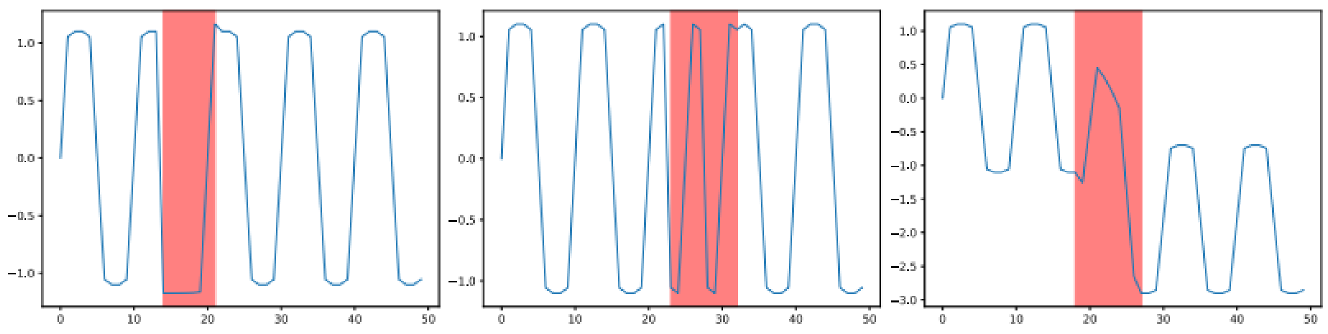

(a) Shapelet Outliers (b) Seasonal Outlier (c) Trend Outlier

Figure 3: *Illustration of three types of pattern outliers.*

3.2.2 Pattern-wise Anomalies

Pattern-wise Anomalies are anomalous subsequence, which are typically discords or in harmonies. There are three major causes of pattern-wise outliers: basic shapelet, seasonality changes and trend alternations. Specifically, given a time series data X, an underlying subsequence $X_{i,j}$ starting from timestamp i to j can be represented by a shapelet function with trend and seasonality:

$$X_{i,j} = \rho\big(2\pi\omega T_{i,j)}\big) + \tau(T_{i,j}) \tag{6}$$

where ρ defines the basic shape of the subsequence ω is the seasonality of the subsequence, τ is the trend function describing overall direction of $X_{i,j}$. By analyzing three components individually, we identify three types of outliers in pattern-wise behaviour, illustrated in Figure 3.

Shapelet Anomalies refer to the subsequence with dissimilar basic shapelets compared with the normal shapelet, which can be defined as:

$$s\big(\rho(.), \widehat{\rho}(.)\big) > \delta \tag{7}$$

where s is a function measures the dissimilarity between two subsequence, such as dynamic time warping $\widehat{\rho}$ is the basic shapelets of expected subsequence, and δ is a threshold.

Seasonal Anomalies are the subsequence with unusual seasonalities compared with the overall seasonality. They have similar basic shapelet and trend but with unusual seasonalities, defined as:

$$s(\omega, \widehat{\omega}) > \delta \tag{8}$$

where $\widehat{\omega}$ is the seasonality of expected subsequence, and δ is a threshold.

Trend Anomalies indicate the subseuqences that significantly alter the trend of the time series, leading to a permanent shift on the mean of the data. This type of outlier retains basic shapelet and seasonality of the normalities but the slope of the trend changes drastically, which can be defined as:

$$s\big(\tau(.), \widehat{\tau}(.)\big) > \delta \tag{9}$$

Where $\widehat{\tau}$ is the trend of normal subsquences, and δ is a threshold.

3.3 Synthesizing Anomalies

In this subsection, we introduce a general and unified synthetic criterion to benchmark the evaluation of different types of Anomalies.

Global Anomalies. Following Equation **4**, we can synthesize a global outlier by letting $\widehat{xt} = \mu(X)$ and $\lambda.\sigma(X)$, i.e., $x_t = \mu(X) \pm \lambda.\sigma(X)$ where $\mu(X)$ denotes the mean, $\sigma(X)$ denotes the standard deviation, and λ controls how much x_t deviates from the expected value.

Contextual Anomalies. Contextual outliers are expected to locally rather than globally deviate from the expected value. Based on Equation **5**, we can set

$\widehat{xt} = \mu\left(X_{t-k,t+k}\right), \delta = \lambda\,.\,\sigma(X_{t-k,t+k})$, i.e., $x_t = \mu\left(X_{t-k,t+k}\right), \pm\ \lambda\,.\,\sigma(X_{t,t+k})$ where μ and σ are instead obtained from a subsequence.

Shapelet Anomalies. Following Equation **7**, we can synthesize a shapelet outlier from timestep i to j by setting ρ to be other shapelets with $X_{i,j} = \rho\left(2\pi\widehat{\omega}T_{i,j)}\right) + \tau(T_{i,j})$ where $\widehat{\omega}$ denotes the expected seasonality, $\hat{\tau}$ denotes the expected trend, and ρ is another shapelet. For instance, we can set ρ ⇢ to be square wave to synthesize a shapelet outlier in a sine wave, illustrated in Figure 3a.

Seasonal Anomalies. Based on Equation 8, we can similarly synthesize a seasonal outlier from timestamp i to j with $X_{i,j} = \hat{\rho}(2\pi\omega\, T_{i,j}) + \hat{\tau}(T_{i,j})$ where ω is another seasonality while $\hat{\rho}$ and $\hat{\tau}$ are the expected ones. Figure 3b gives an example of the seasonal anomaly by setting the seasonality as $2\widehat{\omega}$.

Trend Anomalies. Similarly, we can follow Equation 9 to synthesize the trend outliers with $X_{i,j} = \hat{\rho}\left(2\pi\widehat{\omega}T_{i,j)}\right) + \tau(T_{i,j})$. Figure 3c shows the example with $\tau(T_{i,j})$ = {-1, -2, -3, ..., -($j - i$ +1)}.

4. Methodology

4.1 Isolation and Isolation Forest

the term *isolation* means ‘separating an instance from the rest of the instances. Since anomalies are ‘few and different’ and therefore they are more susceptible to isolation. In a data-induced random tree, partitioning of instances are repeated recursively until all instances are isolated. This random partitioning produces noticeable shorter paths for anomalies since.
(a) the fewer instances of anomalies result in a smaller number of partitions – shorter paths in a tree structure, and
(b) instances with distinguishable attribute-values are more likely to be separated in early partitioning. Hence, when a forest of random trees collectively produces shorter path lengths for some particular points, then they are highly likely to be *anomalies***.**

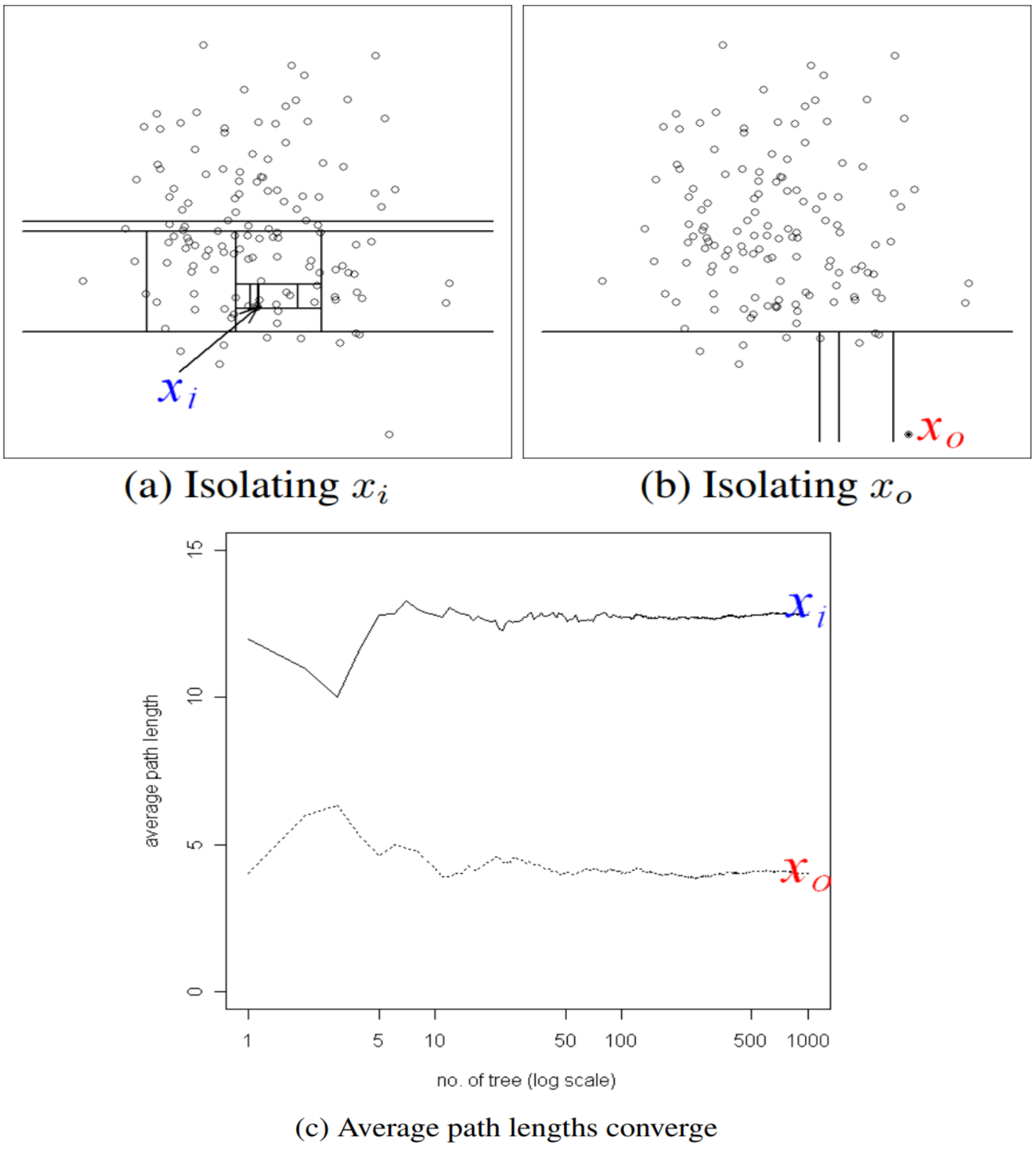


Figure 4. *Anomalies are more susceptible to isolation and hence have short path lengths. Given a Gaussian distribution (135 points), (a) a normal point xi requires twelve random partitions to be isolated; (b) an anomaly xo requires only four partitions to be isolated. (c) averaged path lengths of xi and xo converge when the number of trees increases.*

To demonstrate the idea that anomalies are more susceptible to isolation under random partitioning, we illustrate an example in Figures 4(a) and 4(b) to visualise the random partitioning of a normal point versus an anomaly. We observe that a normal point, x_i, generally requires more partitions to be isolated. The opposite is also true for the anomaly point, x_o, which generally requires less partitions to be isolated. In this example, partitions are generated by randomly selecting an attribute and then randomly selecting a split value between the maximum and minimum values of the selected attribute. Since recursive partitioning can be represented by a tree structure, the number of partitions required to isolate a point is equivalent to the path length from the root node to a terminating node. In this example, the path length of x_i, is greater than the path length of xo. Since each partition is randomly generated, individual trees are generated with different sets of partitions. We average path lengths over a number of trees to find the expected path length. Figure 4(c) shows that the average path lengths of x_o, and x_i, converge when the number of trees increases. Using 1000 trees, the average path lengths of x_o, and x_i, converge to 4.02 and 12.82 respectively. It shows that anomalies are having path lengths shorter than normal instances.

4.2 Isolation Tree.

Let T be a node of an isolation tree. T is either an external-node with no child, or an internal-node with one test and exactly two daughter nodes ($T_{l,} T_r$). A test consists of an attribute q and a split value p such that the test $q < p$ divides data points into $T_{l,}$ and T_r.

Given a sample of data $X = \{ x_{1,\ldots\ldots} x_n \}$of n instances from a d-variate distribution, to build an isolation tree (iTree), we recursively divide X by randomly selecting an attribute q and a split value p, until either:

(i) the tree reaches a height limit,
(ii) $|X| = 1$ or
(iii) all data in X have the same values. *An iTree is a proper binary tree*, where each node in the tree has exactly zero or two daughter nodes.

Assuming all instances are distinct, each instance is isolated to an external node when an iTree is fully grown, in which case the number of external nodes is n and the number of internal nodes is $n - 1$; the total number of nodes of an iTrees is $2n - 1$; and thus, the memory requirement is bounded and only grows linearly with n. The task of anomaly detection is to provide a ranking that reflects the degree of anomaly. Thus, one way to detect anomalies is to sort data points according to their path lengths or anomaly scores; and anomalies are points that are ranked at the top of the list. We define path length and anomaly score as follows.

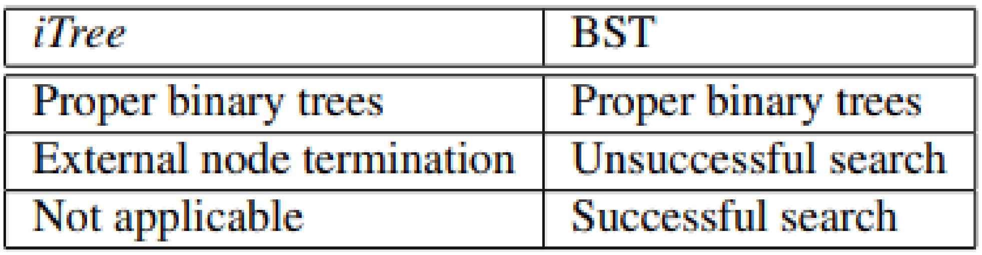
4.2.1 Path Length

h(x) of a point x is measured by the number of edges x traverses an iTree from the root node until the traversal is terminated at an external node.

4.3 An anomaly score

is required for any anomaly detection method. The difficulty in deriving such a score from *h(x)* is that while the maximum possible height of iTree grows in the order of *n*, the average height grows in the order of *log n*. Normalization of *h(x)* by any of the above terms is either not bounded or cannot be directly compared. Since iTrees have an equivalent structure to

Binary Search Tree or BST (see Table 1), the estimation of average *h(x)* for external node terminations is the same as the



iTree	BST
Proper binary trees	Proper binary trees
External node termination	Unsuccessful search
Not applicable	Successful search

Table 1. *List of equivalent structure and operations in iTree and Binary Search Tree (BST)*

unsuccessful search in BST. We borrow the analysis from BST to estimate the average path length of iTree. Given a data set of n instances, Section 10.3.3 of [9] gives the average path length of unsuccessful search in BST as:

$$c(n) = 2H(n - 1) - (2(n - 1)/n), \qquad (10)$$

where *H(i)* is the harmonic number and it can be estimated by $ln(i)$ + 0.5772156649 (Euler's constant). As *c(n)* is the average of *h(x)* given n, we use it to normalise h(x). The anomaly score s of an instance x is defined as:

$$s\ (x, n) = 2^{-E(h(x))/c(n)} \qquad (11)$$

where *E(h(x))* is the average of *h(x)* from a collection of isolation trees. In Equation (11):

- when $E(h(x)) \to c(n)$, $s \to 0.5$;
- when $E(h(x)) \to 0$, $s \to 1$;
- and when $E(h(x)) \to n - 1$, $s \to 0$.

s is monotonic to *h(x)*. Figure 2 illustrates the relationship between *E(h(x))* and *s*, and the following conditions applied where $0 < s \leq 1$ for $0 < h(x) \leq n - 1$. Using the anomaly score *s*, we are able to make the following assessment:

- if instances return s very close to 1, then they are definitely anomalies,
- if instances have *s* much smaller than 0.5, then they are quite safe to be regarded as normal instances, and
- if all the instances return *s* ≈ 0.5, then the entire sample does not really have any distinct anomaly.

A contour of anomaly score can be produced by passing a lattice sample through a collection of isolation trees, facilitating a detailed analysis of the detection result. Figure 5 shows an example of such a contour, allowing a user to visualise and identify anomalies in the instance space. Using the contour, we can clearly identify three points, where $s \geq 0.6$, which are potential anomalies.

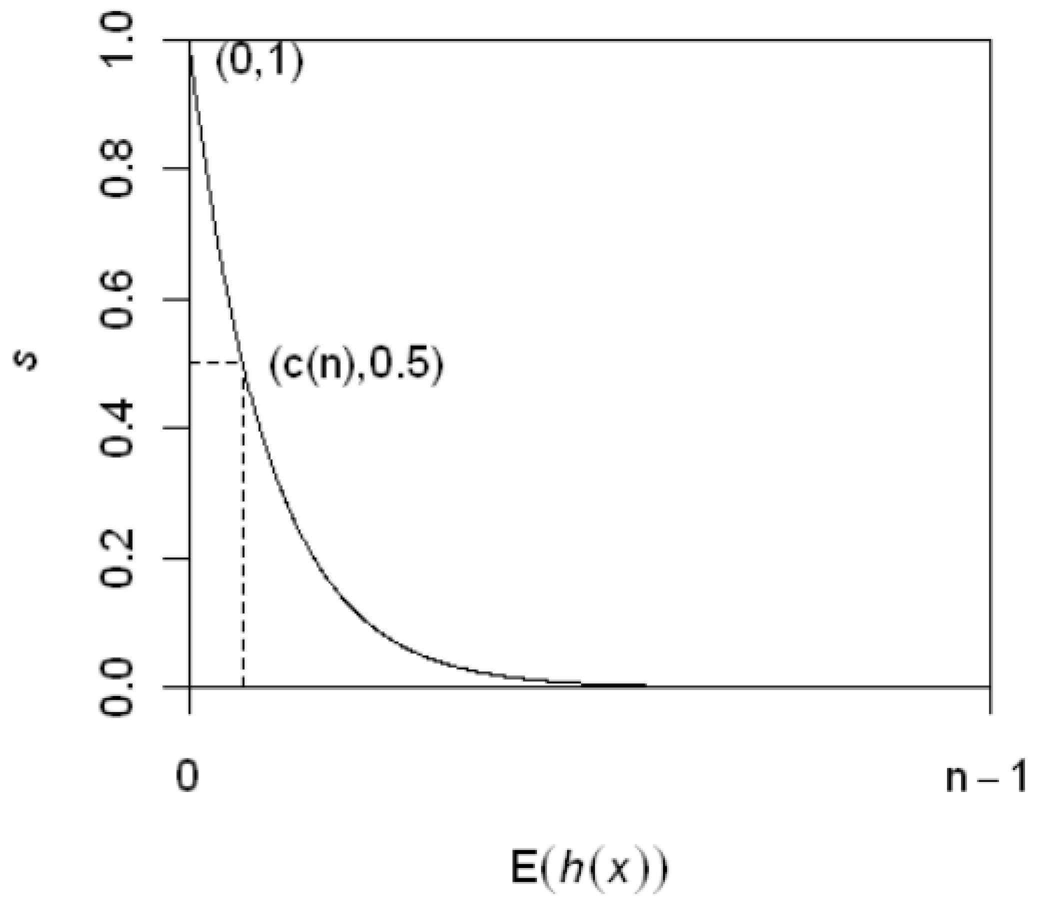


Figure 5*. The relationship of expected path length* ***E(h(x))*** *and anomaly score s.* ***c(n)*** *is the average path length as defined in equation 1. If the expected path length* ***E(h(x))*** *is equal to the average path length* ***c(n),*** *then s = 0.5, regardless of the value of* ***n****.*

5. Characteristic of Isolation Trees

This section describes the characteristic of iTrees and their unique way of handling the effects of swamping and masking. As a tree ensemble that employs isolation trees, iForest

a) identifies anomalies as points having shorter path lengths, and

b) has multiple trees acting as 'experts' to target different anomalies. Since iForest does not need to isolate all of normal instances – the majority of the training sample, *iForest is able to work well with a partial model without isolating all normal points and builds models using a small sample size.*

Contrary to existing methods where large sampling size is more desirable, isolation method works best when the sampling size is kept small. Large sampling size reduces iForest's ability to isolate anomalies as normal instances can interfere with the isolation process and therefore reduces its ability to clearly isolate anomalies. Thus, sub-sampling provides a favourable environment for iForest to work well. Throughout this paper, sub-sampling is conducted by random selection of instances without replacement. Problems of swamping and masking have been studied extensively in anomaly detection [8]. Swamping refers to Figure 3. Anomaly score contour of iForest for a Gaussian distribution of sixty-four points. Contour lines for $s = 0.5, 0.6, 0.7$ are illustrated. Potential anomalies can be identified as points where $s \geq 0.6$. wrongly identifying normal instances as anomalies. When normal instances are too close to anomalies, the number of partitions required to separate anomalies increases – which makes it harder to distinguish anomalies from normal instances. Masking is the existence of too many anomalies concealing their own presence. When an anomaly cluster is large and dense, it also increases the number of partitions to isolate each anomaly. Under these circumstances, evaluations using these trees have longer path lengths making anomalies more difficult to detect. Note that both swamping and masking are a result of too many data for the purpose of anomaly detection. The unique characteristic of isolation trees allows iForest to build a partial model by sub-sampling which incidentally alleviates the effects of swamping and masking. It is because:

1) sub-sampling controls data size, which helps iForest better isolate examples of anomalies and

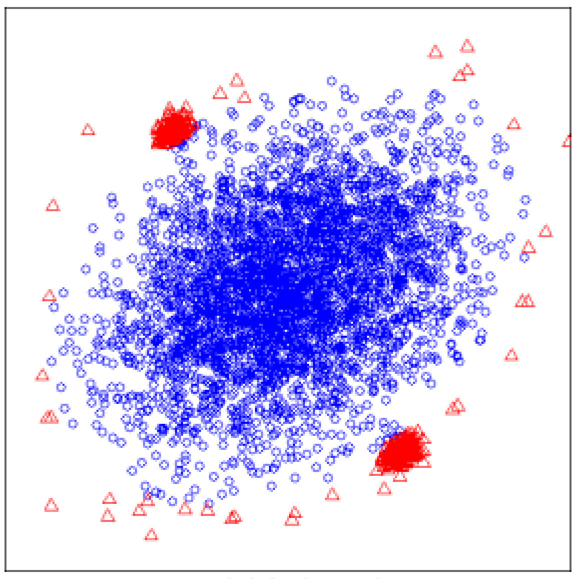
2) each isolation tree can be specialised, as each sub-sample includes different set of anomalies or even no anomaly

Basic Characteristics of Isolation Forest

- it uses **normal samples as the training set** and can allow a few instances of abnormal samples (configurable). You basically feed the algorithm your normal data and it doesn't mind if your dataset is not that well curated, provided you tune the contamination parameter. In other words, **it learns what normal looks like to be able to distinguish the abnormal**,

- it works with the basic assumption that anomalies are **few** and easily **distinguishable,**
- it has a **linear time complexity** with a low constant and a **low memory requirement*,**
- it is **fast** because it doesn't utilize any distance or density measures,
- can **scale up** very well. It seems to work well with high dimensional problems that may have a large number of irrelevant attributes.

To illustrate this, Figure 4(a) shows a data set generated by Mulcross. The data set has two anomaly clusters located close to one large cluster of normal points at the centre. There are interfering normal points surrounding the anomaly clusters, and the anomaly clusters are denser.


(a) Original sample
(4096 instances)

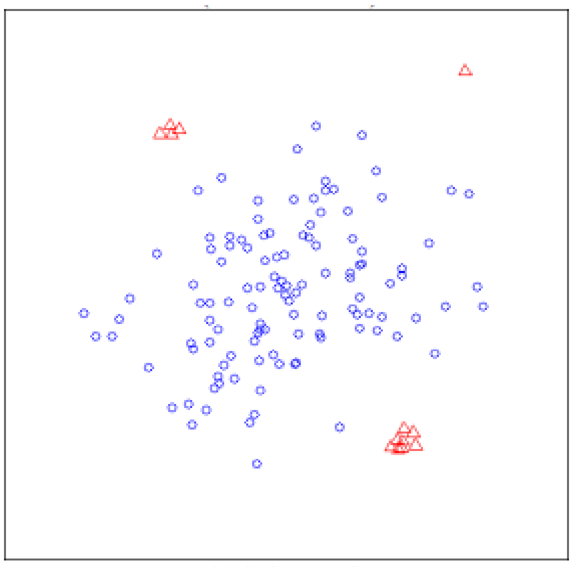

(b) Sub-sample
(128 instances)

Figure 6*. Using generated data to demonstrate the effects of swamping and masking, (a) shows the original data generated by Mul cross. (b) shows a sub-sample of the original data. Circles (◦) denote normal instances and triangles (△) denote anomalies.*

than normal points in this sample of 4096 instances. Figure 6(b) shows a sub-sample of 128 instances of the original data. The anomalies clusters are clearly identifiable in the sub-sample. Those normal instances surrounding the two anomaly clusters have been cleared out, and the size of anomaly clusters becomes smaller which makes them easier to identify. When using the entire sample, iForest reports an AUC of 0.67. When using a sub-sampling size of 128, iForest achieves an AUC of 0.91. The result shows iForest's superior anomaly detection ability in handling the effects swamping and masking through a significantly reduced subsample.

6. Anomaly Detection Using Isolation Forest

Anomaly detection using iForest is a two-stage process. **The first (training)** stage builds isolation trees using subsamples of the training set. **The second (testing)** stage passes the test instances through isolation trees to obtain an anomaly score for each instance

6.1 Training Stage

In the training stage, iTrees are constructed by recursively partitioning the given training set until instances are isolated or a specific tree height is reached of which results a partial model. Note that the tree height limit l is automatically set by the sub-sampling size ψ:

l = *ceiling (log2 ψ),* which is approximately the average tree height [7]. The rationale of growing trees up to the average tree height is that we are only interested in data points that have shorter-than-average path lengths, as those points are more likely to be anomalies. Details of the training stage can be found in Algorithms 1 and 2.

Algorithm 1: *iForest (X, t, ψ)*

Inputs: X - input data, t - number of trees, ψ – subsampling size

Output: a set of t iTrees

```
1. Initialize Forest
2. Set height limit $l$ = ceiling (log2 $\psi$),
3. for $i$ = 1 to $t$ do
4.     $X'$ ← sample ($X$, $\psi$)
5.     Forest ← Forest ∪ iTree ($X'$, 0, $l$)
6. end for
7. return Forest
```

Algorithm 2: *iTree (X, e, l)*

Inputs: X - input data, e - current tree height, l - height limit

Output: an iTree

```
1.  If $e \geq l$ or $|X| \leq 1$ then
2.      return exNode{Size ← $|X|$}
3.  Else
4.      let $Q$ be a list of attributes in $X$
5.      randomly select an attribute $q \in Q$
6.      randomly select a split point p from max and min values of
    attribute $q$ in $X$
7.      $X_l \leftarrow$ filter $(X, q < p)$
8.      $X_r \leftarrow$ filter $(X, q \geq p)$
9.      return inNode {Left t ← iTree $(X_l, e + 1, l)$,
10.                    Right ← iTree $(X_r, e + 1, l)$,
11.                    Split Att ← $q$,
12.                    Split Value ← $p$}
13. End if
```

There are two input parameters to the iForest algorithm. They are the sub-sampling size ψ and the number of trees t. We provide a guide below to select a suitable value for each of the two parameters.

6.1.1 Sub-sampling

size ψ controls the training data size. We find that when ψ increases to a desired value, iForest detects reliably and there is no need to increase ψ further because it increases processing time and memory size without any gain in detection performance. Empirically, we find that setting ψ to 2^8or 256 generally provides enough details to perform anomaly detection across a wide range of data. Unless otherwise specified, we use ψ = 256 as the default value for our experiment. An analysis on the effect sub-sampling size can be found in section 5.2 which shows that the detection performance is near optimal at this default setting and insensitive to a wide range of ψ.

6.1.2 Number of trees

t controls the ensemble size. We find that path lengths usually converge well before t = 100. Unless otherwise specified, we shall use t = 100 as the default value in our experiment. At the end of the training process, a collection of trees is returned and is ready for the evaluation stage. *The complexity of the training an iForest is* $O(t\psi \log \psi)$*.*

6.2 Evaluating Stage

In the evaluating stage, an anomaly score s is derived from the expected path length *E(h(x))* for each test instance. *E(h(x))* are derived by passing instances through each iTree in an iForest. Using Pathlength function, a single path length *h(x)* is derived by counting the number of edges e from the root node to a terminating node as instance x traverses through an iTree. When x is terminated at an external node, where *Size* > *1*, the return value is e plus an adjustment *c(Size)*.

The adjustment accounts for an unbuilt subtree beyond the tree height limit. When *h(x)* is obtained for each tree of the ensemble, an anomaly score is produced by computing *s(x, ψ)* in Equation 11. *The complexity of the evaluation process is O (ntlog ψ),* where n is the testing data size. Details of the pathlength function can be found in Algorithm 3. To find the top *m* anomalies, simply sorts the data using s in descending order. The first m instances are the top *m* anomalies.

Algorithm 3: *iForest (X, t, ψ)*

Inputs: *x* - an instance, *T* - an iTree, *e* - current path length; to be initialized to zero when first called

Output: path length of *x*

```
1. If $T$ is an external node, then
2.     return $e + c\ (T.\ size)$ {$c(.)$ is defined in Equation 1}
3. end if
4. $a \leftarrow T.\ splitAtt$
5. if $x_a < T.\ splitValue$ then
6.     return $PathLength\ (x, T.\ left, e + 1)$
7. else {$x_a \geq T.\ split\ Value$}
8.     return $PathLength\ (x, T.\ right, e + 1)$
9. end if
```

7 Complexity Analysis of Isolation Forest Algorithm

7.1 Training Stage

To derive the time complexity of the Isolation Forest algorithm in the training stage, we need to consider the construction of individual isolation trees (iTrees). Let's use the following variables:

- n: The number of instances in the training set.
- t: The number of trees in the Isolation Forest ensemble.
- ψ: The sub-sampling size, representing the number of instances randomly selected for each tree.
- l: The tree height limit, which is approximately the average tree height.

7.1.1 Best Case Time Complexity:

In the best case, the iTrees are constructed efficiently, with minimal partitioning required to isolate instances. The best-case time complexity occurs when the instances are evenly distributed among the trees, and each tree grows up to the height limit l.

- *Number of iTrees (t)*: In the best case, all t trees are constructed with the same height limit l.
- *Time Complexity per Tree:* For each tree, the partitioning process (Algorithm 1) involves selecting a random subset of instances of size ψ, and then recursively partitioning the instances until either they are isolated or the height limit l is reached. In the best case, the height limit l is reached in $log2\ \psi$ steps. The time complexity per tree is $O(log2\ \psi)$.

  Best Case Time Complexity = $O(t * log2\ \psi)$

7.1.2 Worst Case Time Complexity:

In the worst case, the iTrees require maximum partitioning to reach the height limit l. This occurs when the instances are clustered in a way that necessitates a deep partitioning process for each tree.

- *Number of iTrees (t):* In the worst case, all t trees are constructed with the same height limit l.
- Time Complexity per Tree: For each tree, the partitioning process (Algorithm 1) involves selecting a random subset of instances of size ψ, and then

recursively partitioning the instances until either they are isolated or the height limit l is reached. In the worst case, the height limit l is reached in *log2 ψ* steps. The time complexity per tree is *O(log2 ψ).*

*Worst Case Time Complexity = O(t * log2 ψ)*

7.1.3 Impact of Input Parameters:

1. **Sub-sampling *Size* (ψ):** The sub-sampling size determines the number of instances randomly selected for each tree. Larger values of ψ generally lead to shorter average tree heights (l), reducing the construction time of individual trees. However, very large values of ψ can increase memory usage and computational overhead in the evaluation stage.

2. **Number of Trees (t):** The number of trees (t) in the Isolation Forest ensemble has a direct impact on the overall training time. Increasing the number of trees enhances the algorithm's performance but also increases the training time linearly.

In practice, it is essential to find an appropriate balance between *ψ* and *t* to achieve efficient training without sacrificing anomaly detection accuracy. Additionally, selecting a suitable *ψ* that aligns with the dataset's size and characteristics can help optimize the overall time complexity of the training stage.

7.2 Evaluating Stage:

To derive the time complexity of the Isolation Forest algorithm in the evaluating stage, we will consider two cases: the best case and the worst case. Let's use the following variables:

- n: The number of instances in the testing data.
- t: The number of trees in the Isolation Forest ensemble.
- ψ: The average path length of an instance in an isolation tree.
- m: The number of top anomalies to be detected (m <= n).

7.2.1 Best Case Time Complexity:

In the best case, each instance in the testing data follows the shortest path through every isolation tree (i.e., the height of each tree is minimized). The time complexity for computing the anomaly scores in the evaluating stage is determined by the time it takes to evaluate a single tree.

For each instance, the path length function (Algorithm 3) needs to be computed for each of the t trees. Thus, the time complexity for the best case is given by:

Best Case Time Complexity = O(t)

7.2.2 Worst Case Time Complexity:

In the worst case, each instance follows the longest path through every isolation tree (i.e., the tree is fully grown, and no early termination occurs). The time complexity for computing the anomaly scores in the evaluating stage is determined by the time it takes to evaluate a single tree.

For each instance, the path length function (Algorithm 3) needs to be computed for each of the t trees. As the maximum height of an isolation tree is $log(\psi)$, the time complexity for the worst case is given by:

*Worst Case Time Complexity = O(t * log(ψ))*

7.2.3 Impact of Input Parameters:

1. *Number of Trees (t):* The number of trees in the Isolation Forest ensemble *(t)* has a direct impact on the time complexity. Increasing the number of trees

improves the anomaly detection accuracy but also increases the computational overhead in the evaluating stage.

2. *Average Path Length (ψ):* The average path length (ψ) represents the typical depth an instance traverses through a single isolation tree. A larger value of ψ allows more instances to traverse deeper into trees, potentially leading to a longer evaluating time.
3. *Testing Data Size (n):* The size of the testing data (n) affects the overall time complexity linearly. As the testing data increases, the evaluating stage's time complexity grows proportionally.
4. *Number of Top Anomalies (m):* The number of top anomalies to be detected (m) does not directly affect the time complexity of the evaluating stage. However, the time complexity for sorting the data based on anomaly scores can impact the overall performance when selecting the top m instances.

8. Resources and Data Analysis

8.1 Libraries Used:

1. **NumPy and Pandas:** *NumPy* is an open-source Python library that facilitates efficient numerical operations on large quantities of data. There are a few functions that exist in NumPy that we use on pandas Data Frames. For us, the most important part about NumPy is that pandas is built on top of it. So, NumPy is a dependency of Pandas.

   Pandas is a very popular library for working with data (its goal is to be the most powerful and flexible open-source tool, and in our opinion, it has reached that goal). DataFrames are at the center of pandas. A DataFrame is structured like a table or spreadsheet. The rows and the columns both have indexes, and you can perform operations on rows or columns separately.
   A pandas DataFrame can be easily changed and manipulated. Pandas has helpful functions for handling missing data, performing operations on columns and rows, and transforming data.

2. **Matplotlib:** Matplotlib is a low-level graph plotting library in python that serves as a visualization utility. Matplotlib is open source and we can use it freely. Matplotlib is mostly written in python, a few segments are written in C, Objective-C and JavaScript for Platform compatibility.

3. **HoloViews:** HoloViews is an open-source Python library designed to make data analysis and visualization seamless and simple. With HoloViews, you can usually express what you want to do in very few lines of code, letting you focus on what you are trying to explore and convey, not on the process of plotting.

4. **Seaborn:** Seaborn is an amazing visualization library for statistical graphics plotting in Python. It provides beautiful default styles and color palettes to make statistical plots more attractive. It is built on top matplotlib library and is also closely integrated with the data structures from pandas. Seaborn aims to make visualization the central part of exploring and understanding data. It provides dataset-oriented APIs so that we can switch between different visual representations for the same variables for a better understanding of the dataset.

5. **Azure Anomaly Detector SDK:** Anomaly Detector is an AI service with a set of APIs, which enables you to monitor and detect anomalies in your time series data with little ML knowledge, either batch validation or real-time inference. The Anomaly Detector API enables you to monitor and find abnormalities in your time series data by automatically identifying and applying the correct statistical models, regardless of industry, scenario, or data volume.

6. **Scipy and Scikit-learn:** SciPy is a comprehensive library for scientific and numerical computing in Python. It provides a wide range of modules and functions for tasks such as linear algebra, optimization, signal processing, statistics, and more. It is designed to handle mathematical computations and scientific data analysis, offering tools for numerical integration, interpolation, Fourier transforms, and linear algebra operations. On the other hand, scikit-learn is a machine learning library that focuses specifically on providing tools and algorithms for data pre-processing, supervised and unsupervised learning, model evaluation, and predictive analytics. It includes

modules for tasks like classification, regression, clustering, dimensionality reduction, and model selection.

scikit-learn provides a diverse range of machine learning algorithms, including linear regression, logistic regression, decision trees, random forests, SVM, and k-means clustering. Its consistent interface enables effortless experimentation and performance evaluation of various models. In contrast, SciPy lacks built-in machine learning algorithms but offers essential tools for scientific computing, which can complement scikit-learn. However, it does not provide the same dedicated machine-learning functionality as scikit-learn.

7. **Ensemble Library:** the goal of **ensemble methods** is to combine the predictions of several base estimators built with a given learning algorithm in order to improve generalizability / robustness over a single estimator.

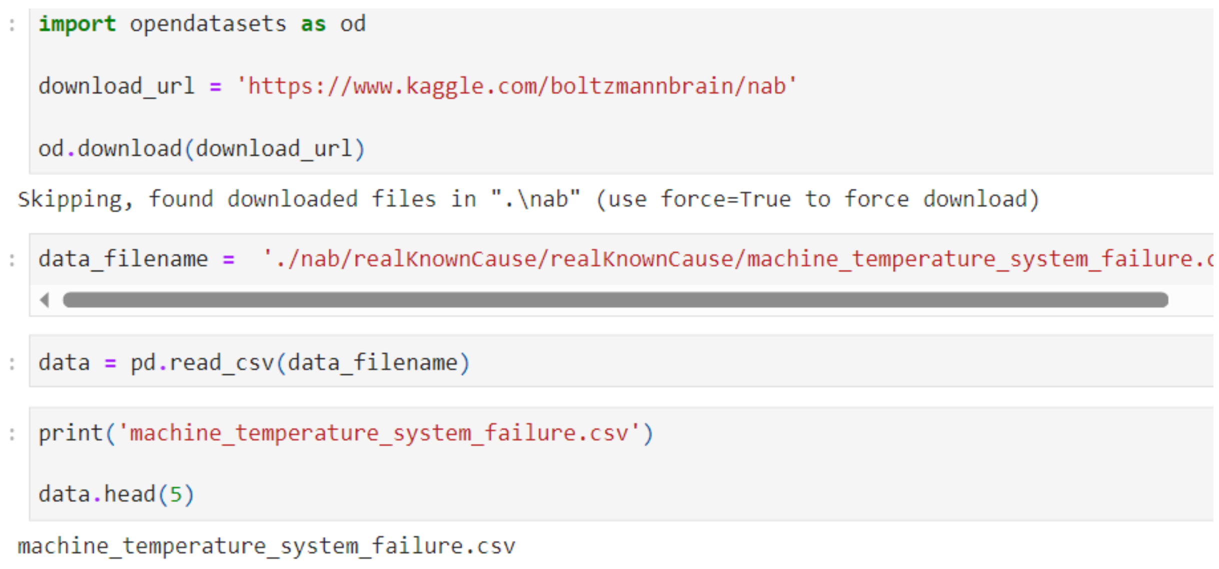
8. **Tabulate and IPython:** tabulate is a module that allows you to display table data beautifully. It is not part of standard Python library.

    - list of lists (in general case - iterable of iterables)
    - dictionary list (or any other iterable object with dictionaries). Keys are used as column names
    - dictionary with iterable objects. Keys are used as column names.

8.2 Data Loading and Pre-processing:

As above, we use 'machine_temperature_system_failure.csv' for our analysis. According to dataset information, it has the following features:

- Temperature sensor data of an internal component of a large, industrial machine.
- The first anomaly is a planned shutdown of the machine.
- The second anomaly is difficult to detect and directly led to the third anomaly, a catastrophic failure of the machine.



```
import opendatasets as od

download_url = 'https://www.kaggle.com/boltzmannbrain/nab'

od.download(download_url)
```

```
Skipping, found downloaded files in ".\nab" (use force=True to force download)
```

```
data_filename =  './nab/realKnownCause/realKnownCause/machine_temperature_system_failure.c
```

```
data = pd.read_csv(data_filename)
```

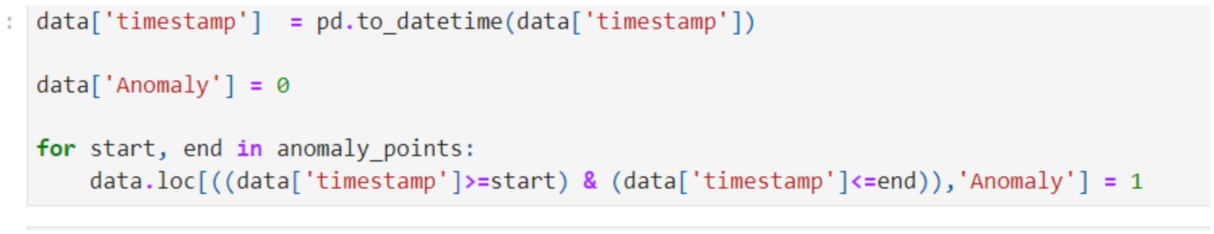
```
print('machine_temperature_system_failure.csv')

data.head(5)
```

```
machine_temperature_system_failure.csv
```

pre-processing:

- augmenting the original data with an additional column called 'Anomaly', which will indicate whether each data point falls within the detected anomaly periods specified in the anomaly_points list. Here's a breakdown of what the code is doing:



```
data['timestamp']  = pd.to_datetime(data['timestamp'])

data['Anomaly'] = 0

for start, end in anomaly_points:
    data.loc[((data['timestamp']>=start) & (data['timestamp']<=end)),'Anomaly'] = 1
```

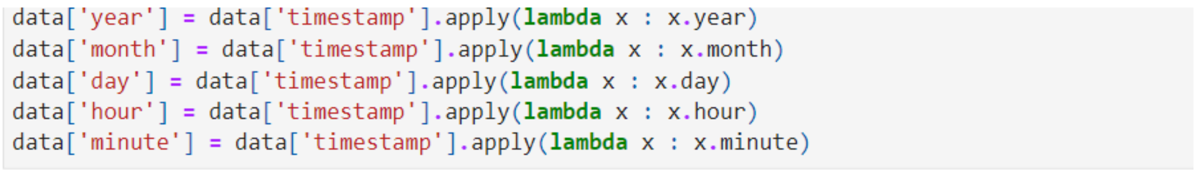
1. **data['timestamp'] = pd.to_datetime(data['timestamp'])**: This line converts the 'timestamp' column in the 'data' DataFrame to pandas datetime format. This step ensures that the timestamps are in a standardized format, making it easier to perform temporal operations.
2. **data['Anomaly'] = 0**: A new column named 'Anomaly' is added to the 'data' DataFrame, initialized with 0 for all rows. This column will be used to mark data points that fall within the detected anomaly periods.
3. The following **for** loop iterates through the **anomaly_points** list:

   a. **for start, end in anomaly_points:** This loop unpacks each sublist from the **anomaly_points** list, assigning the start timestamp to the variable 'start' and the end timestamp to the variable 'end'.

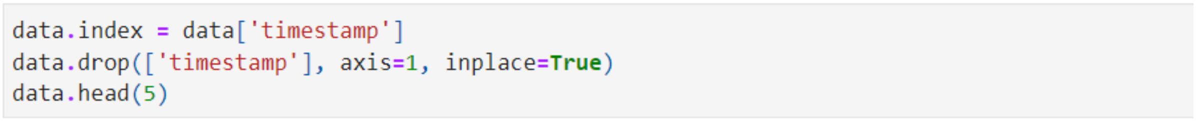
   b. **data.loc[((data['timestamp'] >= start) & (data['timestamp'] <= end)), 'Anomaly'] = 1**: For each detected anomaly period specified by 'start' and 'end', this line sets the 'Anomaly' column to 1 for all rows in the 'data' DataFrame where the 'timestamp' falls within the specified anomaly period. This effectively marks those rows as anomalies.

After running the code, the 'Anomaly' column in the 'data' DataFrame will contain binary values (0 or 1), where 1 indicates that the corresponding data point falls within a detected anomaly period specified in the **anomaly_points** list.

- preprocessing the 'data' DataFrame by extracting various time-related features from the 'timestamp' column and then reorganizing the DataFrame to use the 'timestamp' as the new index. Let's break down what each line of the code is doing:



```
data['year'] = data['timestamp'].apply(lambda x : x.year)
data['month'] = data['timestamp'].apply(lambda x : x.month)
data['day'] = data['timestamp'].apply(lambda x : x.day)
data['hour'] = data['timestamp'].apply(lambda x : x.hour)
data['minute'] = data['timestamp'].apply(lambda x : x.minute)
```



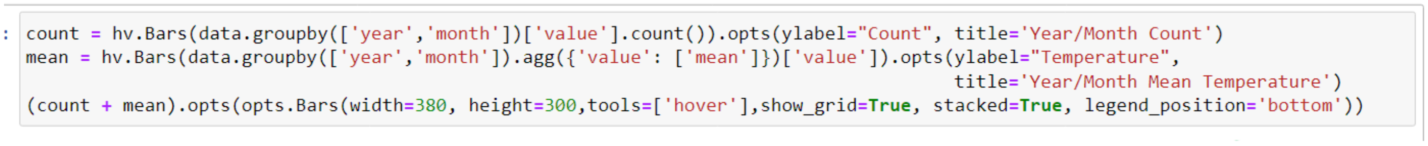
```
data.index = data['timestamp']
data.drop(['timestamp'], axis=1, inplace=True)
data.head(5)
```

1. **data['year'] = data['timestamp'].apply(lambda x : x.year)**: This line creates a new column called 'year' in the 'data' DataFrame. It extracts the year component from each 'timestamp' using the **apply** method with a lambda function. The 'year' column will contain the year values corresponding to each timestamp.
2. **data['month'] = data['timestamp'].apply(lambda x : x.month)**: This line creates a new column called 'month' in the 'data' DataFrame. It extracts the month component from each 'timestamp' using the **apply** method with a lambda function. The 'month' column will contain the month values corresponding to each timestamp.
3. **data['day'] = data['timestamp'].apply(lambda x : x.day)**: This line creates a new column called 'day' in the 'data' DataFrame. It extracts the day component from each 'timestamp' using the **apply** method with a lambda function. The 'day' column will contain the day values corresponding to each timestamp.
4. **data['hour'] = data['timestamp'].apply(lambda x : x.hour)**: This line creates a new column called 'hour' in the 'data' DataFrame. It extracts the hour component from each 'timestamp' using the **apply** method with a lambda function. The 'hour' column will contain the hour values corresponding to each timestamp.
5. **data['minute'] = data['timestamp'].apply(lambda x : x.minute)**: This line creates a new column called 'minute' in the 'data' DataFrame. It extracts the minute component from each 'timestamp' using the **apply** method with a lambda function. The 'minute' column will contain the minute values corresponding to each timestamp.
6. **data.index = data['timestamp']**: This line sets the 'timestamp' column as the new index of the 'data' DataFrame. By doing this, the 'timestamp' will now be the primary identifier for each row in the DataFrame.
7. **data.drop(['timestamp'], axis=1, inplace=True)**: This line drops the original 'timestamp' column from the DataFrame since its information has already been extracted into separate columns ('year', 'month', 'day', 'hour', 'minute'). The **inplace=True** argument ensures that the DataFrame is modified in place without returning a new DataFrame.

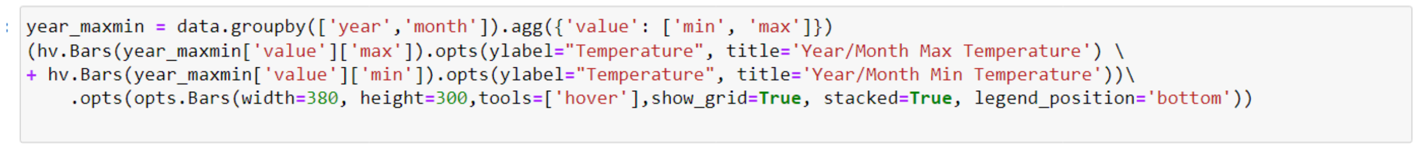
After running the code, the 'data' DataFrame will have additional columns ('year', 'month', 'day', 'hour', 'minute') representing the corresponding components of the 'timestamp' in a more accessible format.

8.3 EDA (Exploratory Data Analysis):

- Doing Analysis of data various plots using the Holoviews library to visualize different aspects of the 'data' DataFrame. Let's break down each plot and its purpose:



```python
count = hv.Bars(data.groupby(['year','month'])['value'].count()).opts(ylabel="Count", title='Year/Month Count')
mean = hv.Bars(data.groupby(['year','month']).agg({'value': ['mean']})['value']).opts(ylabel="Temperature",
                                                                                    title='Year/Month Mean Temperature')
(count + mean).opts(opts.Bars(width=380, height=300,tools=['hover'],show_grid=True, stacked=True, legend_position='bottom'))
```



```python
year_maxmin = data.groupby(['year','month']).agg({'value': ['min', 'max']})
(hv.Bars(year_maxmin['value']['max']).opts(ylabel="Temperature", title='Year/Month Max Temperature') \
+ hv.Bars(year_maxmin['value']['min']).opts(ylabel="Temperature", title='Year/Month Min Temperature'))\
    .opts(opts.Bars(width=380, height=300,tools=['hover'],show_grid=True, stacked=True, legend_position='bottom'))
```

- code creates various Holoviews plots for visualizing the 'data' DataFrame. Here's a short description of each plot:
    1. **Count and Mean Temperature Plot (Count + Mean):**
        - This plot combines the count of data points and the mean temperature for each year and month.
        - The count bars show the distribution of data points, and the mean temperature bars represent the average temperature for each month.
        - The plots are stacked vertically with a legend at the bottom.
    2. **Max and Min Temperature Plot (Max + Min):**
        - This plot combines the maximum and minimum recorded temperatures for each year and month.
        - The maximum temperature bars show the highest recorded temperature, and the minimum temperature bars show the lowest recorded temperature for each month.
        - The plots are stacked vertically with a legend at the bottom.

The code effectively creates these visualizations in a concise manner using Holoviews, making it easier to understand the distribution and characteristics of temperature data across different months and years.

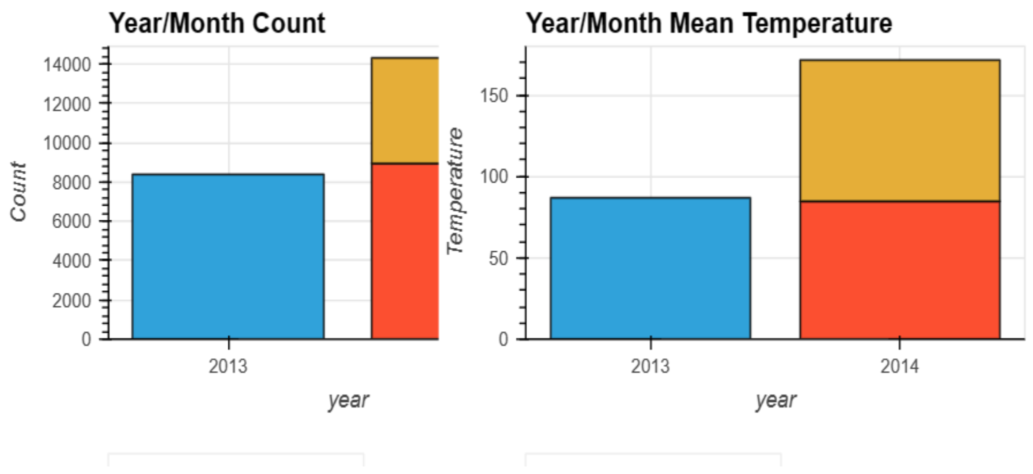


Figure 7 *Count and Mean Temperature Plot (Count + Mean)*

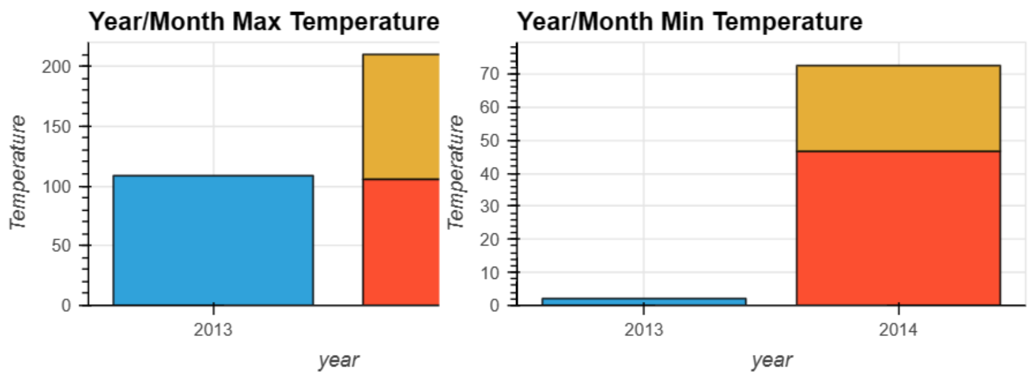


Figure 8 *Max and Min Temperature Plot (Max + Min):*

Holoviews Distribution plot to visualize the distribution of temperature values in the 'data' DataFrame. Here's a brief description of the plot:

```
hv.Distribution(data['value']).opts(opts.Distribution(title="Temperature Distribution",
xlabel="Temperature", ylabel="Density", width=700, height=300,tools=['hover'],show_grid=True))
```

Temperature Distribution Plot:

- The plot represents the distribution of temperature values in the 'data' DataFrame.
- The x-axis is labeled as "Temperature," and the y-axis is labeled as "Density."
- The plot displays the density of data points at various temperature values, showing how frequently each temperature value occurs.
- The plot has a title "Temperature Distribution" and dimensions of width=700 and height=300.
- Interactive tools, such as hover, are enabled for exploring data points.
- Grid lines are displayed to aid in reading the plot.

This Distribution plot provides an overview of how temperature values are distributed in the dataset, helping identify any patterns, outliers, or characteristics in the temperature data.

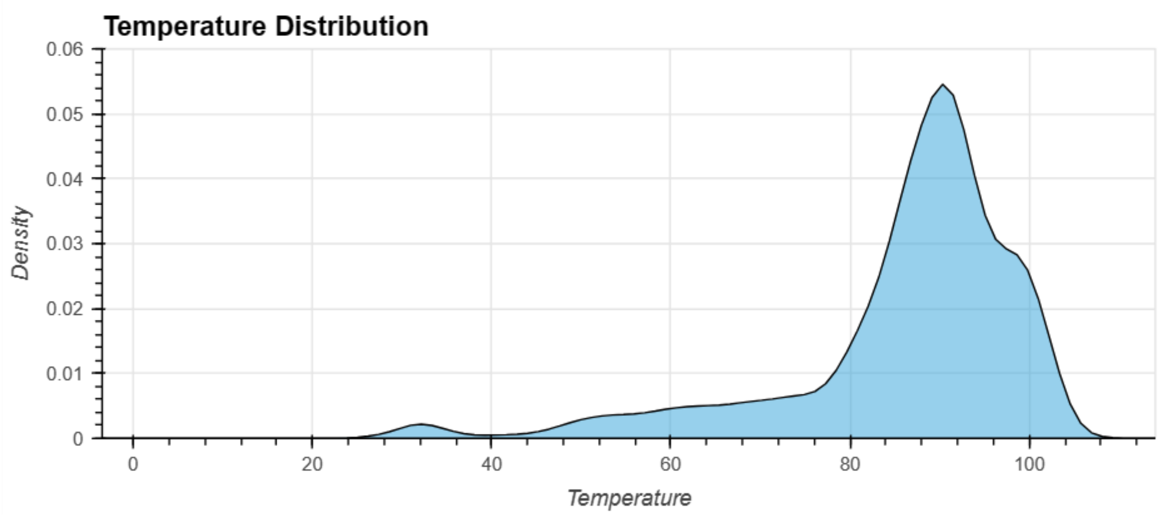
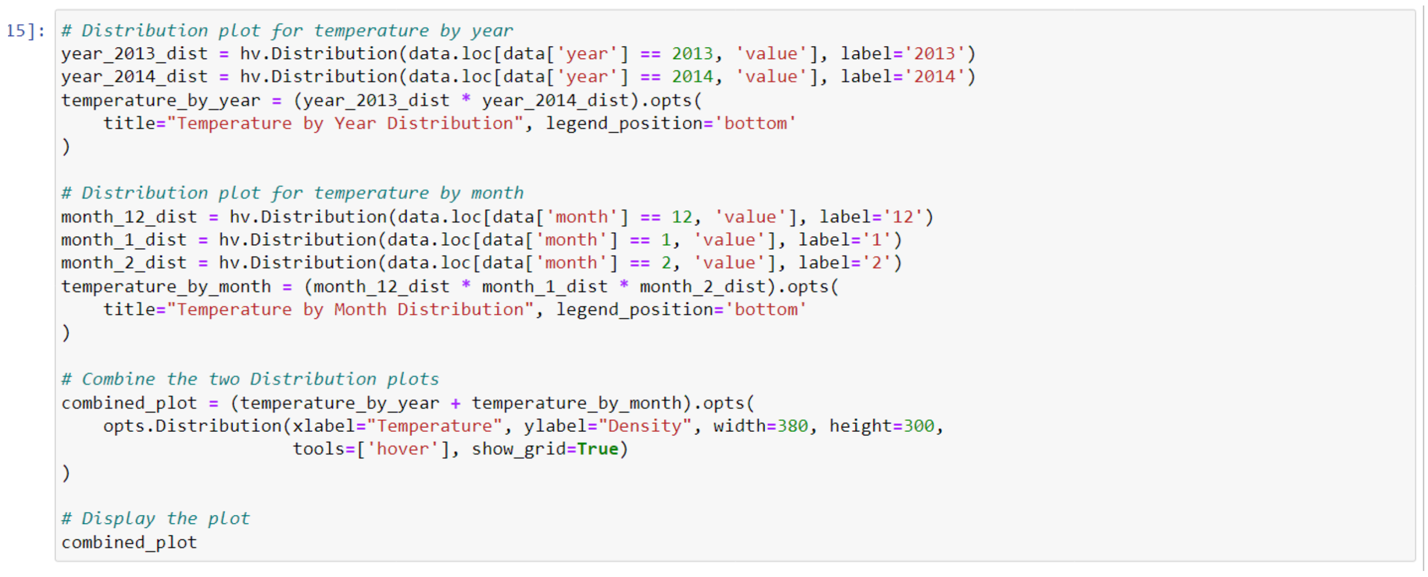


Figure 9 *Temperature Distribution Plot*

- creates two Holoviews Distribution plots to visualize the distribution of temperature values in the 'data' DataFrame based on the year and month. Here's a brief description of each plot and how they are combined:



```
15]: # Distribution plot for temperature by year
     year_2013_dist = hv.Distribution(data.loc[data['year'] == 2013, 'value'], label='2013')
     year_2014_dist = hv.Distribution(data.loc[data['year'] == 2014, 'value'], label='2014')
     temperature_by_year = (year_2013_dist * year_2014_dist).opts(
         title="Temperature by Year Distribution", legend_position='bottom'
     )

     # Distribution plot for temperature by month
     month_12_dist = hv.Distribution(data.loc[data['month'] == 12, 'value'], label='12')
     month_1_dist = hv.Distribution(data.loc[data['month'] == 1, 'value'], label='1')
     month_2_dist = hv.Distribution(data.loc[data['month'] == 2, 'value'], label='2')
     temperature_by_month = (month_12_dist * month_1_dist * month_2_dist).opts(
         title="Temperature by Month Distribution", legend_position='bottom'
     )

     # Combine the two Distribution plots
     combined_plot = (temperature_by_year + temperature_by_month).opts(
         opts.Distribution(xlabel="Temperature", ylabel="Density", width=380, height=300,
                           tools=['hover'], show_grid=True)
     )

     # Display the plot
     combined_plot
```

Temperature by Year Distribution Plot:

- The plot represents the distribution of temperature values for the years 2013 and 2014.
- The legend indicates different colors for each year.
- The x-axis is labeled as "Temperature," and the y-axis is labeled as "Density."
- The plot's title is "Temperature by Year Distribution," and the legend is positioned at the bottom.
- The plot compares the temperature distributions between the two years.

Temperature by Month Distribution Plot:

- The plot represents the distribution of temperature values for the months of December (month 12), January (month 1), and February (month 2).
- The legend indicates different colors for each month.
- The x-axis is labeled as "Temperature," and the y-axis is labeled as "Density."
- The plot's title is "Temperature by Month Distribution," and the legend is positioned at the bottom.
- The plot compares the temperature distributions among the three selected months.

Combined Plot:

- The combined plot merges the "Temperature by Year Distribution" plot and the "Temperature by Month Distribution" plot side by side.
- Each plot shows the distribution of temperature values with different colors indicating the respective year or month.
- The plot has dimensions of width=380 and height=300, and interactive tools (e.g., hover) are enabled for exploration.
- Grid lines are displayed to aid in reading the plots.

This combination of plots provides a comprehensive visualization of temperature distributions across years and selected months, enabling easy comparisons and insights into the temperature data's characteristics during different time periods.

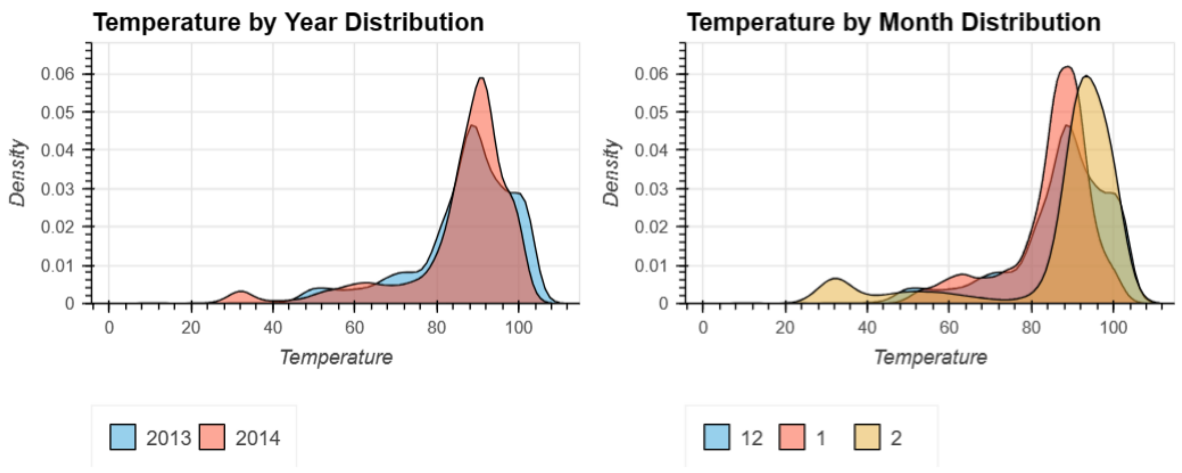


Figure 10 *Temperature by Month/year Distribution Plot*

8.4 Time Series Analysis:

- The code uses Holoviews to plot the temperature data as a curve and highlights anomaly points in red. Anomalies are obtained from the 'data' DataFrame, where 'Anomaly' column equals 1. The resulting plot shows temperature over time with red points indicating anomalies.

```
##plot temperature & its given anomaly points.
anomalies = [[ind, value] for ind, value in zip(data[data['Anomaly']==1].index, data.loc[data['Anomaly']==1,'value'])]
(hv.Curve(data['value'], label="Temperature") * hv.Points(anomalies, label="Anomaly Points").opts(color='red',
                                    legend_position='bottom', size=2, title="Temperature & Given Anomaly Points"))\
    .opts(opts.Curve(xlabel="Time", ylabel="Temperature", width=700, height=400,tools=['hover'],show_grid=True))
```

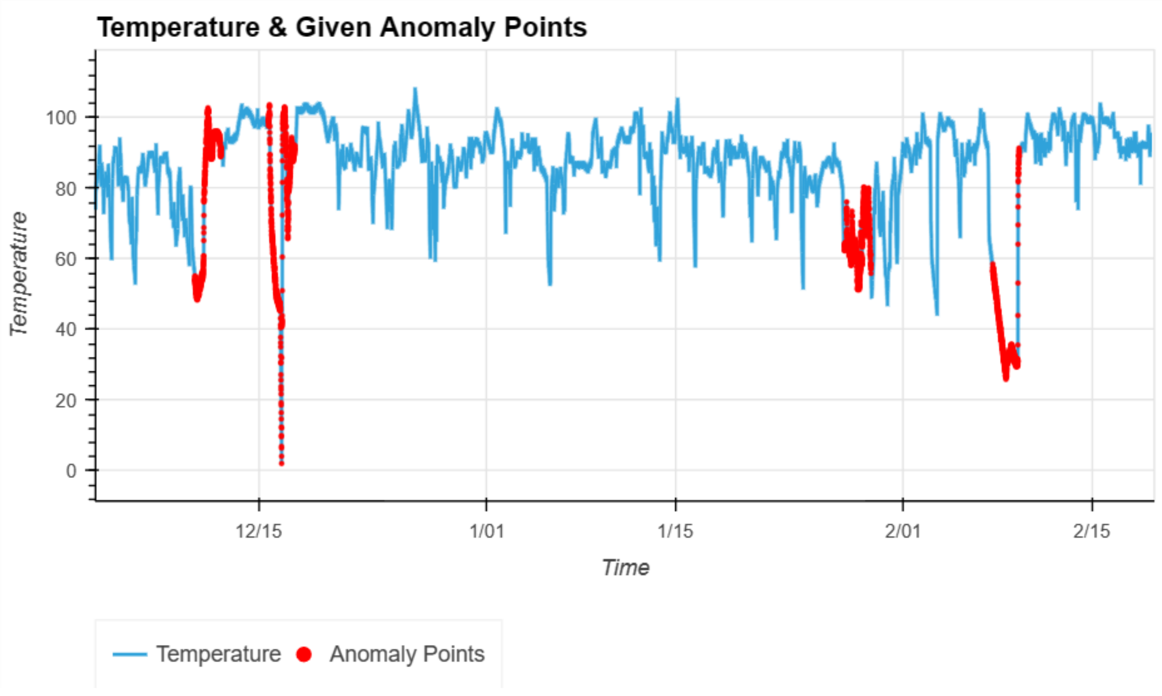


Figure 11 *Temperature with given Anomalies Distribution Plot*

- The plot represents the daily mean temperature trends over time. The x-axis represents time, which is days. The y-axis represents the average temperature for each day. The curve shows how the daily mean temperature varies over the given time period.
- With this plot, you can observe the overall temperature patterns, such as seasonal changes or long-term trends in the daily mean temperature. The curve's fluctuations may indicate daily temperature variations or significant events affecting the temperature on specific days. It allows you to gain insights into the average temperature behavior on a daily basis.

```
hv.Curve(data['value'].resample('D').mean()).opts(opts.Curve(title="Temperature Mean by Day", xlabel="Time",
ylabel="Temperature", width=700, height=300,tools=['hover'],show_grid=True))
```

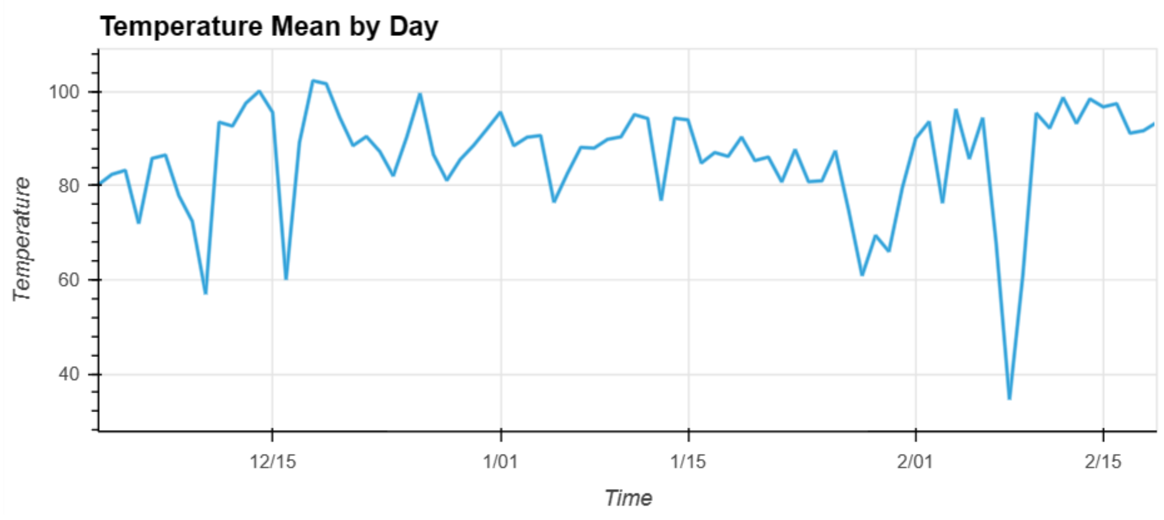


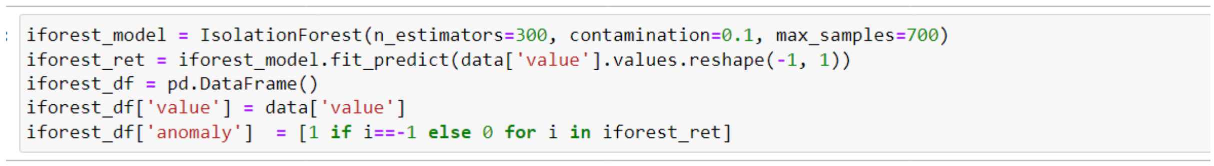
Figure 12 *Temperature mean by day Distribution Plot*

9.Model Implementation

I implemented four distinct models in model implementation and then assessed them using various evaluating matrices. The models I implemented include isolation forest, LOF, one class SVM, and microsoft anomaly detection API.

9.1 Isolation Forest: I have already covered the math and reasoning behind the working and implementation of the isolation forest; here we will simply address how the model is finally implemented, as well as an explanation of the code and its detected Anomaly Output.

The provided code performs anomaly detection using the Isolation Forest algorithm on the 'data' DataFrame, and creates a new DataFrame 'iforest_df' to store the results. Here's a brief description of each step:



```
iforest_model = IsolationForest(n_estimators=300, contamination=0.1, max_samples=700)
iforest_ret = iforest_model.fit_predict(data['value'].values.reshape(-1, 1))
iforest_df = pd.DataFrame()
iforest_df['value'] = data['value']
iforest_df['anomaly']  = [1 if i==-1 else 0 for i in iforest_ret]
```

1. Isolation Forest Model (iforest_model):

- The Isolation Forest algorithm is used to create an anomaly detection model.
- The `**n_estimators**` parameter is set to 300, specifying the number of isolation trees in the forest.
- The `**contamination**` parameter is set to 0.1, indicating the proportion of outliers or anomalies expected in the data.
- The `**max_samples**` parameter is set to 700, defining the maximum number of samples used to build each isolation tree.

2. Model Fitting and Prediction (iforest_ret):

- The model is fitted to the 'value' column of the 'data' DataFrame.
- The `**fit_predict**` method is used to perform anomaly prediction on the 'value' column.
- The result is stored in '**iforest_ret**', which is an array containing 1 for inliers (normal data points) and -1 for outliers (anomalies).

3. Create 'iforest_df' DataFrame (iforest_df):

- A new DataFrame named '**iforest_df**' is created to store the 'value' column and the anomaly results.
- The 'value' column is filled with the original 'value' column from the 'data' DataFrame.
- The 'anomaly' column is created using list comprehension, where it is set to 1 for anomalies (when the corresponding value in '**iforest_ret**' is -1) and 0 for inliers.

After executing the code, '**iforest_df**' will contain the 'value' column with temperature values and the 'anomaly' column, indicating whether each data point is classified as an anomaly (1) or an inlier (0) by the Isolation Forest algorithm. This DataFrame is further analyzed or visualized to identify and understand the detected anomalies in the temperature data.

code generates a Holoviews visualization to display the Isolation Forest results, highlighting the detected anomalies in the 'iforest_df' DataFrame. Here's a brief description of the plot:

```
: anomalies = [[ind, value] for ind, value in zip(iforest_df[iforest_df['anomaly']==1].index,
                                                  iforest_df.loc[iforest_df['anomaly']==1,'value'])]
  (hv.Curve(iforest_df['value'], label="Temperature") * hv.Points(anomalies, label="Detected Points")
   .opts(color='red', legend_position='bottom', size=2, title="Isolation Forest - Detected Points"))\
      .opts(opts.Curve(xlabel="Time", ylabel="Temperature", width=700, height=400,tools=['hover'],show_grid=True))
```

Isolation Forest - Detected Points Plot:

- The plot consists of two elements combined together using the '*' operator.
- The first element, 'hv.Curve(iforest_df['value'], label="Temperature")', represents a curve plot of the temperature values over time.
- The second element, 'hv.Points(anomalies, label="Detected Points")', represents red points on the curve plot, indicating the detected anomalies.

Options and Settings:

- The 'hv.Curve' represents the temperature curve over time, and the 'hv.Points' represents the anomalies detected by the Isolation Forest.
- The 'color' option is set to 'red' to display the anomalies in red on the curve plot.
- The legend is positioned at the bottom.
- The plot has dimensions of width=700 and height=400, and interactive tools (e.g., hover) are enabled for exploration.
- Grid lines are displayed to aid in reading the plot.

This combined plot allows you to visualize the temperature values over time as a curve and simultaneously identifies the detected anomalies as red points on the curve.

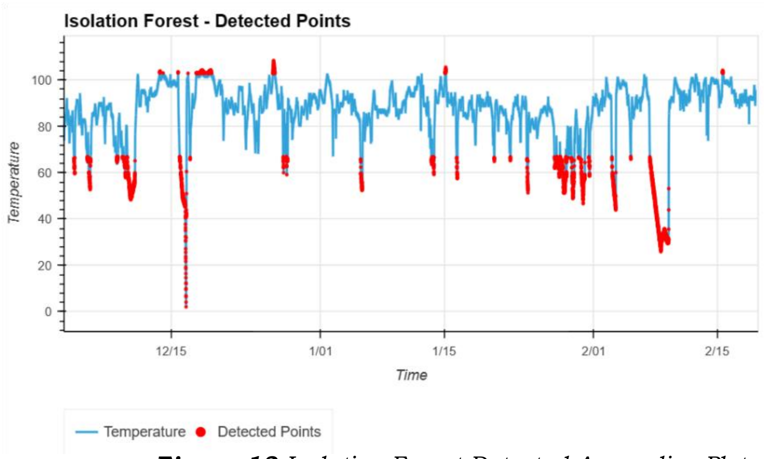


Figure 13 *Isolation Forest Detected Anomalies Plot*

9.2 Microsoft Anomaly Detector: In this section we will discuss about the Microsoft anomaly detection service how it works and how do we have implemented this model.

At Microsoft, they develop a time-series anomaly detection service which helps customers to monitor the time-series continuously and alert for potential incidents on time.

They proposed a novel algorithm based on Spectral Residual (SR) and Convolutional Neural Network (CNN). Our work is the first attempt to borrow the SR model from visual saliency detection domain to time-series anomaly detection. Moreover, they innovatively combine SR and CNN together to improve the performance of SR model.

9.2.2 System Overview:

The system consists of three key components: **data ingestion, experimentation platform, and online compute**. Users register monitoring tasks by ingesting time-series data from various sources. The ingestion worker updates time-series data at designated intervals, and points are stored in a time-series database via Kafka. The online compute module processes incoming data points using Flink, performing anomaly detection for each time-series. The smart alert processor correlates anomalies from multiple time-series to generate incident reports. An experimentation platform evaluates anomaly detection models through offline experiments and online A/B tests. The platform allows users to label anomalies for model evaluation. Effective models are deployed to production via Azure machine learning service and K8s.

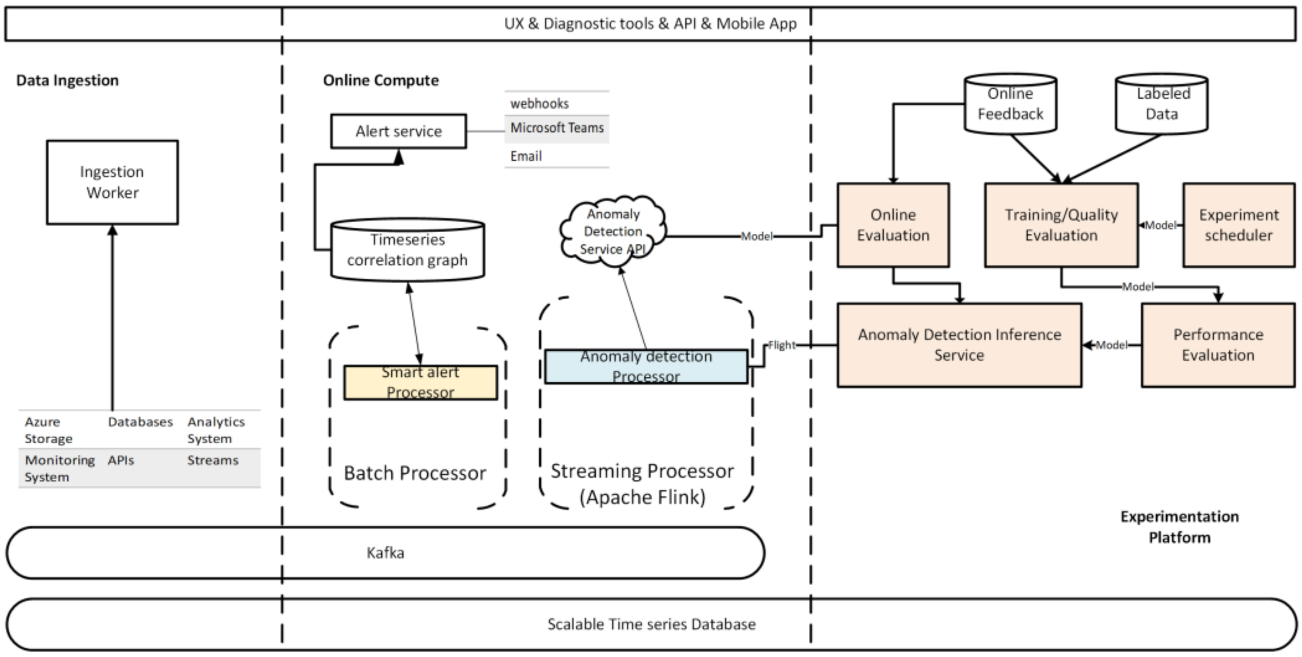


Figure 14 *Detailed diagram of the system overview of Detector*

9.2.3 Methodology:

we propose a novel method, SR-CNN, which applies CNN on the output of SR model directly. CNN is responsible to learn a discriminate rule to replace the single threshold adopted by the original SR solution. The problem becomes much easier to learn the CNN model on SR results than on the original input sequence. Specifically, we can use artificially generated anomaly labels to train the CNN-based discriminator. In the following sub-sections, we introduce the details of SR and SR-CNN methods respectively.

9.2.3.1 SR (Spectral Residual)

The Spectral Residual (SR) algorithm consists of three major steps: (1) Fourier Transform to get the log amplitude spectrum; (2) calculation of spectral residual; and (3) Inverse Fourier Transform that transforms the sequence back to spatial domain. Mathematically, given a sequence x, we have

$$A(f) = Amplitude(\mathcal{F}(\mathrm{x})) \quad (1)$$

$$P(f) = phrase(\mathcal{F}(\mathrm{x}) \quad (2)$$

$$L(f) = log(A(f)) \quad (3)$$

$$AL(f) = h_q(f) \cdot L(f) \quad (4)$$

$$R(f) = L(f) - AL(f) \quad (5)$$

$$S(x) = \left|\left|\mathcal{F}^{-1}\left(\exp\left(R(f) + iP(f)\right)\right)\right|\right| \quad (6)$$