Anomaly Detection for Water Quality Data

Anomaly Detection for Water Quality Data

By Yan Yan,

A Thesis Submitted to the School of Graduate Studies in the Partial Fulfillment of the Requirements for the Degree Master of Science

McMaster University
Master of Science (2019)
Hamilton, Ontario (Computing and Software)

TITLE: Anomaly Detection for Water Quality Data
AUTHOR: Yan Yan (McMaster University)
SUPERVISORS: Dr. John Copp, Dr. Emil Sekerinski
NUMBER OF PAGES: x, 123

Abstract

Real-time water quality monitoring using automated systems with sensors is becoming increasingly common, which enables and demands timely identification of unexpected values. Technical issues create anomalies, which at the rate of incoming data can prevent the manual detection of problematic data.

This thesis deals with the problem of anomaly detection for water quality data using machine learning and statistic learning approaches. Anomalies in data can cause serious problems in posterior analysis and lead to poor decisions or incorrect conclusions. Five time series anomaly detection techniques: local outlier factor (machine learning), isolation forest (machine learning), robust random cut forest (machine learning), seasonal hybrid extreme studentized deviate (statistic learning approach), and exponential moving average (statistic learning approach) have been analyzed. Extensive experimental analysis of those techniques have been performed on data sets collected from sensors deployed in a wastewater treatment plant.

The results are very promising. In the experiments, three approaches successfully detected anomalies in the ammonia data set. With the temperature data set, the local outlier factor successfully detected all twenty-six outliers whereas the seasonal hybrid extreme studentized deviate only detected one anomaly point. The exponential moving average identified ten time ranges with anomalies. Eight of them cover a total of fourteen anomalies. The reproducible experiments demonstrate that local outlier factor is a feasible approach for detecting anomalies in water quality data. Isolation forest and robust random cut forest also rate high anomaly scores for the anomalies. The result of the primary experiment confirms that local outlier factor is much faster than isolation forest, robust random cut forest, seasonal hybrid extreme studentized deviate and exponential moving average.

Acknowledgements

Foremost, I would like to express my sincere gratitude to my advisor Prof. Emil Sekerinski for the continuous support of my study and research.

I would like to thank Dr. John Copp from Primodal and other team members for their support, feedback, cooperation which helped me get results of better quality.

Contents

List of Figures

List of Tables

Declaration of Authorship

I, Yan YAN, declare that this thesis titled, "Anomaly Detection for Water Quality Data" and the work presented in it are my own. Where I have consulted the work of others, this is always clearly stated.

Chapter 1

Introduction

1.1 Introduction

Water makes up more than 70% of the surface of the Earth. It is essential to all the aspects of our lives and all life needs water to survive. Monitoring the quality of water bodies can help to protect the quality of that water. It can be difficult to measure the quality of a water body. For example, water is a vast network of linked parts such as rivers, creeks, wetlands, estuaries, and lakes. Furthermore, assessing water quality may be complicated if these linked parts contain different levels of pollution.

1.2 Water Quality Monitoring

Water quality is used to describe the chemical, physical, biological and other content of water that change with the seasons and geographic areas. It is affected by environmental influences and human activities. Geological, hydrological and climate are the most important environmental influences that affect both the quantity and the quality of available water. On the other hand, the effects of human activities on water quality are widespread and vary in the degree to which they disrupt the ecosystem and restrict water use (Jamie Bartram and Ballance 1996). Water quality issues influence both human and environmental health, thus the more our water is monitored the more likely contamination problems will be recognized and prevented.

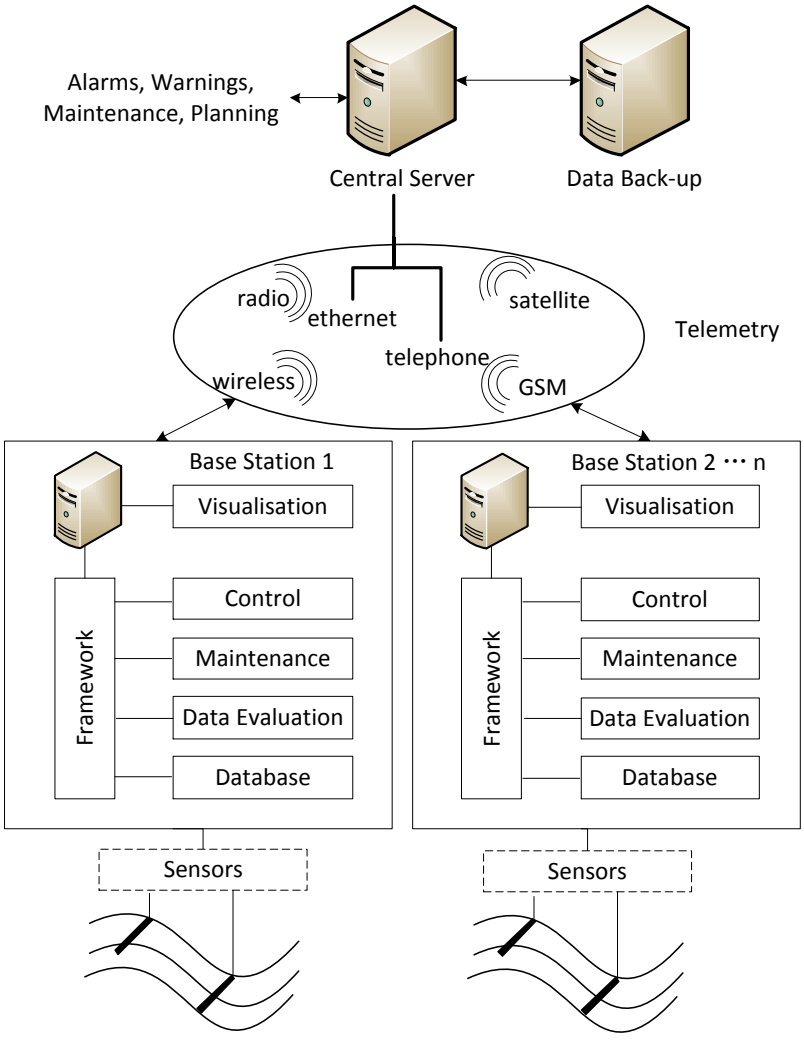
Anomalies (Section 2.3.4) in water quality data can cause serious problems in posterior analysis and lead to false decisions or incorrect conclusions. Anomalies are caused by many things:

1. They may be caused by ecological phenomena like rainfall or floods. These anomalies are expected.
2. They may arise from both human and technical errors which are unexpected. For instance:
    - Communication errors happen between server and sensors.
    - The senor probe is dirty.
    - Senor is pulled out of water for cleaning.
    - Equipment is malfunctioning.

Collaborating with Primodal System, this research focused on possible approaches to automatically detect anomalies in time series water quality data.

Various new technologies have been developed for remote autonomous sensing of water systems. In cooperation with Primodal Systems, an RSM30 Water Monitoring System has been deployed on site at the Dundas wastewater treatment plant in Hamilton, Ontario. This system aimed to monitor temperature, ammonia, potassium and chloride in the primary effluent.

1.3 RSM30 Water Monitoring System



Source: (Primodal 2013)
FIGURE 1.1: Structure of RSM30

The RSM30 water monitoring system is a high-tech system used to monitor real-time water quality data. Built upon the combination of highly customized software and cutting-edge technology, the RSM30 is a user-friendly and a flexible waterside monitoring station.

The RSM30 is a high-level monitoring system which has been designed to perform all types of monitoring tasks. And it has the ability to avoid the errors that other systems have experienced. The RSM30 is so flexible that it can simultaneously execute different tasks at different measurement locations. Furthermore, it fully supports multiple transmission protocols between sensor and base station. As a consequence, it allows users to choose the most suitable sensor for the application very conveniently.

The RSM30 provides different telemetry options and a proactive maintenance feature with very low consumption of energy. Therefore, the monitoring network can be used remotely as well. More importantly, it supports advanced evaluation of the data quality. The risk of storing erroneous data can be largely reduced by real-time evaluation of data quality performed by RSM30.

The RSM30 combines hardware with comprehensive software (PrecisionNow) that logs and analyses the data for errors and problems in real-time. Flexibility in the software means that users can easily expand and customize the analysis to suit the specific system under study.

1.3.1 PrecisionNow Software

PrecisionNow is a software designed and developed by Primodal Systems to run the RSM30. It controls both the storage and analysis of the measurement data. It manages communication with the sensors and information flows to the central server. As the culmination of continuous development and testing, PrecisionNow is not just intuitive and fluid, but powerful and flexible. It is the perfect mix of easy to use combined with the flexibility of a fully customizable solution.

1.4 Anomaly Detection in Water Quality Data

Detecting anomalous data also known as outlier detection, refers to the process of finding patterns in data that do not conform to the expected behavior (Chandola, Banerjee, and Kumar 2009). Anomaly detection is under study by countless research groups to resolve problems in various application domains and several techniques have been specifically developed. Anomaly detection is very important because many applications use the collected data as significant and actionable information and therefore data quality is paramount. For instance, an anomalous traffic pattern in a computer network could mean that a hacked computer is sending out sensitive data to an unauthorized destination (Ertoz, Lazarevic, E. Eilertson, Tan, Dokas, Kumar, and Srivastava 2003). An anomalous MRI image may indicate the presence of malignant tumors (Spence, Parra, and Sajda 2001) or anomalies in credit card transaction data could indicate credit card or identity theft (Aleskerov, Freisleben, and Rao 1997).

Anomaly detection algorithms use models of normal behaviors to automatically detect deviations. One of the major benefits of anomaly detection algorithms is their capability to detect unforeseen changes.

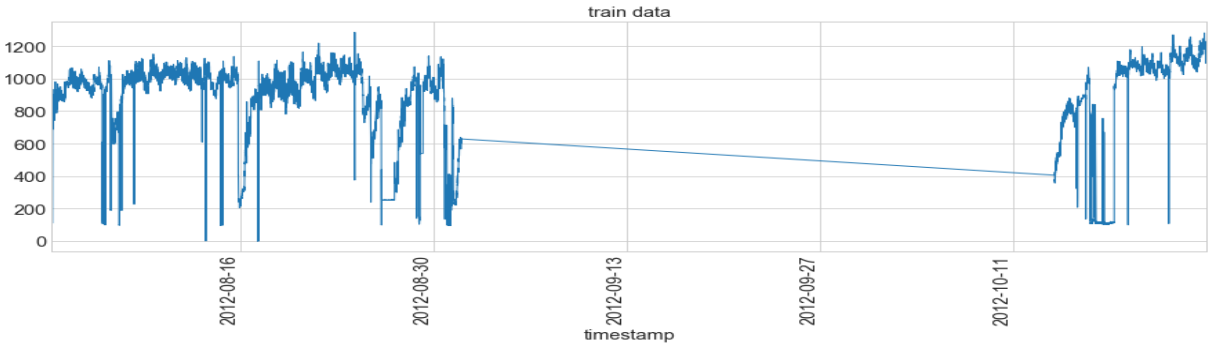


FIGURE 1.2: Anomalous Time Series

In water quality monitoring systems, temporal data is collected by various kinds of sensors. Theoretically, abnormal data can be detected by distinguishing outliers in the time series corresponding to the historical data. This research investigates the problem of anomaly detection in univariate time series. In statistics analysis, univariate time series data is a single set of values over time.

1.5 Case Study

The research involved the analysis of water quality data in order to determine the trends in signals with the hopes of creating software capable of spotting outliers and validating the data's accuracy.

The following is the system deployment information:

- Location: Data was collected using an RSM30 located at the Dundas wastewater treatment facility, Hamilton, Ontario, Canada.
- Probe: IQ Sensor Net Varion@ Plus 700 IQ
- Sensors: ammonia, potassium, chloride, temperature
- Time Interval: 5 minutes and 1 minute

1.6 Reproducible Research

Reproducible research has been used to describe the concept that publications of academic research, and more generally, scientific claims, are published with their full computational environment, raw data, notebook and software code (Fomel and Claerbout 2009).

Reproducible research enables other researchers to reproduce the results, verify the findings and possibly build upon previous work. However, one must bear in mind that even though a study can be reproduced, the conclusions or claims still could be wrong. The reproducibility might be the only thing that a researcher can guarantee about a study.

1.6.1 Jupyter Notebook

Jupyter Notebook is a publishing format for reproducible computational workflows. It is an open-source web application on which users can create and share documents that contain both computer code (such as Python) and rich text elements (like equations or figures). It can be used for data cleaning and transformation, numerical simulation, statistical modeling, data visualization, machine learning (Jupyter 2019).

Jupyter Notebook allows users to execute code in a browser. It provides various pluggable kernels to support more than 40 different programming languages like Python, R, C++, Java and Scala. It can be shared with others using the Jupyter Notebook Viewer as well as Dropbox and GitHub. Jupyter Notebook can display the computation output using rich media representations such as HTML, images and LaTeX. More importantly, Jupyter Notebook also leverages big data tools, such as Apache Spark, from Python, R and Scala and users can explore that same data with Pandas, Scikit-learn, ggplot2, TensorFlow (Jupyter 2019).

Until recently, most scientific research has reproducibility problems as they are simply summarized in scientific papers without any original calculations, and even worse, the raw data might be lost eventually. Jupyter Notebook is a tool to support the reproducibility in science as it can be accessed by anyone to reproduce the final result by following the same calculations and more importantly, user can easily interact with the original experimental data.

1.6.2 Resources

This thesis is based on the concept of reproducible research using Jupyter Notebook. The Notebooks for these experiments are included in this thesis in Section 3.3 and Section 3.4 in Chapter 3. The following are the links to the resources which have been used for this thesis.

- The project repository is hosted on Gitlab including the test data and Jupyter Notebook.

  https://gitlab.cas.mcmaster.ca/waterquality/reports/tree/master/YanYan19/Experiments

- Anaconda is used to setup the environment for experiments. It installs the Python development environment with standard libraries and Jupyter Notebook.

  https://www.anaconda.com

- Jupyter is used to create the shareable documents for experiments with embedded Python code, visualizations and explanatory markdown.

  https://jupyter.org

- The programming language for experiment is Python 3.6.

  https://www.python.org

- Pandas, the Python data analysis library, is used for data retrieving and processing in Notebook.

  https://pandas.pydata.org

- Matplotlib, the Python plotting library, is used to generate figures in Notebook.

https://matplotlib.org

- Python machine learning library Scikit-learn has been used in evaluation experiments (Chapter 3). Specifically, Local Outlier Factor (Section 2.3.5) in Scikit-learn is used to detect anomaly. Scikit-learn is a free open source Python library for machine learning and it provides a lot of simple and efficient tools for data mining as well as data analysis. Dedicate documentation and numerous examples are available on its official website.

  https://scikit-learn.org

- Twitter AnomalyDetection is an open-source R package to detect anomalies which is used in evaluation experiments (Section 2.4.1).

  https://github.com/twitter/AnomalyDetection

- LinkedIn Luminol is a light weight python library for time series data analysis which is used in evaluation experiments (Section 2.4.3).

  https://github.com/linkedin/luminol

- Package eif is a Python library for the extended isolation forest algorithm (Section 2.3.7).

  https://github.com/sahandha/eif

- Package rrcf is a Python implementation of the robust random cut forest algorithm for anomaly detection proposed by (Guha, Mishra, Roy, and Schrijvers 2016) (Section 2.3.8).

  https://github.com/kLabUM/rrcf

Chapter 2

Anomaly Detection Techniques

2.1 Related Work

Intrusion detection system is a valuable security tool because it can detect new unknown cyber attacks. Many computers are vulnerable to attack because of bad security practices, for example, security software is not installed or updated as needed. In response to corporate cyber attacks, (Auskalnis, Paulauskas, and Baskys 2018) created an application based on Local Outlier Factor algorithm to detect anomalies in computer networks.

In 2015 Twitter launched AnomalyDetection to detect anomalies by means of statistics. AnomalyDetection is an open-source R package. It was designed to detect both global and local anomalies and is very robust if data presents seasonality and trends. This package can be used in various applications and has, for example, been used to detect spikes in user engagement, find spam, or resolve issues in financial engineering. Although used with time series, the package can also be used to detect anomalies in a vector of numerical values (Twitter 2015).

LinkedIn also created a lightweight python library, Luminol, for temporal data analysis in 2015. The two major functionalities Luminol supports are anomaly detection and correlation and it can be used to investigate possible causes of anomalies (LinkedIn 2015).

2.2 Techniques Comparison

In this section, a brief overview of popular techniques is given for anomaly detection and then discuss three chosen techniques in detail in the following sections.

Figure 2.1 shows popular techniques for anomaly detection.

Nearest neighbors based anomaly detection techniques are based on local density and built on the k-nearest neighbors algorithm which is introduced in Section 2.9. Generally speaking, they are based on the assumption that the normal data instances sit around a dense neighborhood whereas the anomalies are far away. The nearest data instances are evaluated using scores which are measured by distance. Local outlier factor (LOF) is one of the density-based anomaly detection algorithms, and it is built on a distance metric called reachability distance (Section 2.3.5). Connectivity-based outlier factor (COF) (Tang, Chen, Fu, and Cheung 2002), local outlier probability (LoOP) (Kriegel, Kröger, Schubert, and Zimek 2009), influenced outlierness

(INFLO) (Jin, Tung, Han, and Wang 2006) and local correlation integral (LOCI) are popular nearest neighbors based anomaly detection algorithms.

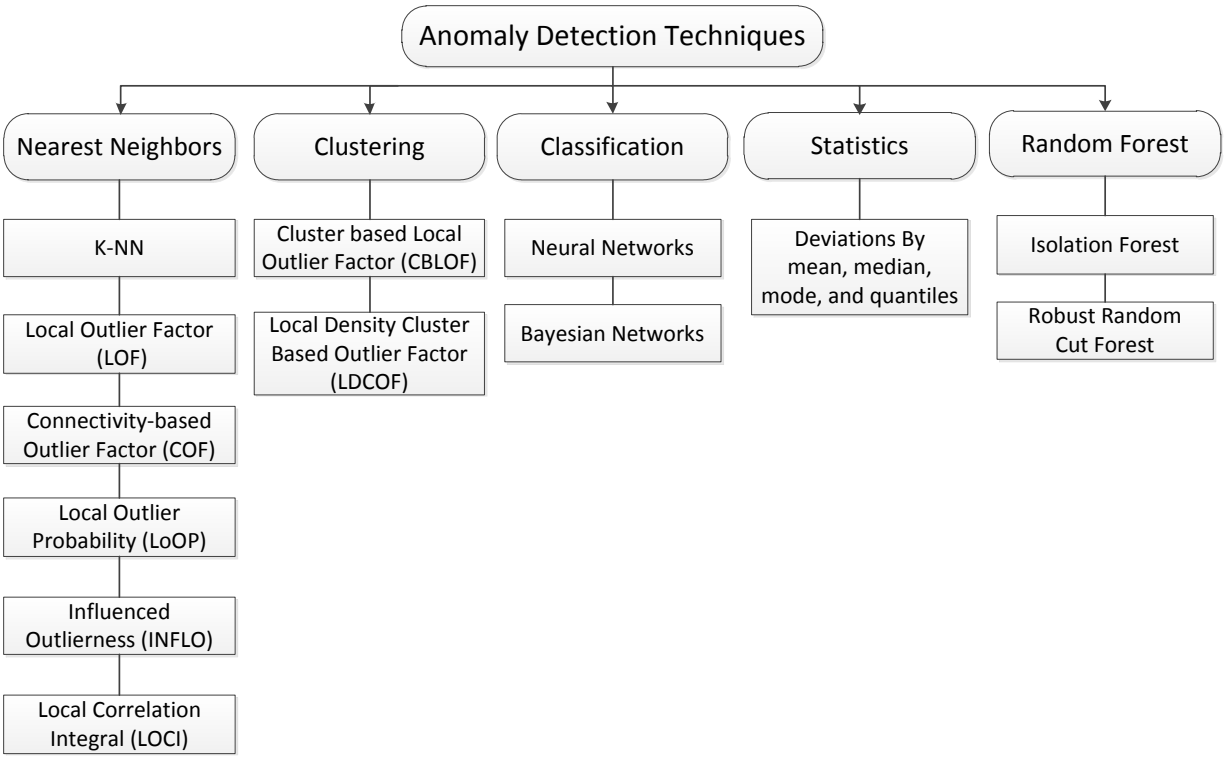


FIGURE 2.1: Anomaly Detection Techniques

Clustering based anomaly detection is unsupervised learning (Section 2.4). These algorithms assume that objects that are similar tend to belong to similar groups (clusters) and the similarity is determined by distance.

For anomaly detection, clustering based anomaly detection techniques assume that anomalies are in sparse and small clusters or not in any cluster. In practice, k-means is widely used to cluster the data in a first step due to the low computational complexity. Cluster-based local outlier factor (CBLOF) uses clustering to determine dense areas and then performs a density estimation for each cluster to identify the anomalies (He, Xu, and Deng 2003). The local density cluster-based outlier factor (LDCOF) estimates the densities of each cluster by assuming a spherical distribution of the cluster members (Amer and Goldstein 2012).

Classification is a technique to categorize data into different classes with labels. There are only two distinct classes for anomaly detection as normal class and abnormal class. The support vector machine (SVM, also refers to support-vector network) is a supervised machine learning algorithm (Section 2.3) for two-group classification problems. The machine conceptually implements the following idea: input vectors are non-linearly mapped to a very high-dimension feature space (Cortes and Vapnik 1995). Bayesian nonlinear SVM was developed by (Wenzel, Galy-Fajou, Deutsch, and Kloft 2017) which enables the application of Bayesian SVMs to big data for classification. Artificial neural network (ANN) is an information processing systems

that is inspired by biological neural networks. An autoencoder is a type of artificial neural network used to learn efficient data coding in an unsupervised manner (Liou, Cheng, Liou, and Liou 2014). The autoencoder network is trained on a data set that represents the normal operating state. It monitors the re-construction errors and uses a probability distribution to classify if a data point is normal or not.

Statistical methods usually calculate deviations from common statistical properties of a distribution and flag the abnormal points with deviations above a threshold. Mean, median, mode, and quantiles are commonly used properties for such calculation. For example, the definition of an anomaly can be defined as the one that deviates beyond a certain threshold from the mean.

Random forests or random decision forests is a learning algorithm for classification, regression and other tasks that operates by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes (classification) or mean prediction (regression) of the individual trees (Ho 1995).

Table 2.1 lists the advantages and disadvantages of these common algorithms. Local Outlier Factor, Seasonal Hybrid Extreme Studentized Deviate and Exponential Moving Average are used in this research mainly because:

- Water quality data is time series data with only one variable and is not suitable for clustering.
- Classification based algorithms like neural networks and SVM require model training with labeled data. It is difficult to train a generic model for classification and different models may generate different results for the same data point.
- LOF is based on K-NN and other nearest neighbors-based algorithms are extensions of LOF.

Algorithms	Advantages	Disadvantages
K-Nearest Neighbors	• Simple to implement • Easy to understand • No training involved	• Large storage requirements • High computational complexity • Sensitive to distance function
Local Outlier Factor	• Well-known and good algorithm for contextual anomaly detection	• Relies on its direct neighborhood • Performs poorly with many point anomalies
Clustering based algorithms	• Suitable for many types of data • Low computational complexity	• Effectiveness depends on the clustering method • May fail if normal data does not create any clusters
Support vector Machine (SVM)	• Find the best separation hyperplane • Deal with high dimensional data • Can learn elaborate concepts	• Require both positive and negative examples • Require lots of memory • Some numerical stability problems • Need to select a good kernel function
Neural Networks	• Fulfil tasks that a linear program cannot • Does not need to be reprogrammed for each learning	• Needs training to operate • Requires high processing time for large neural networks • The architecture needs to be emulated
Isolation Forest	• Explicitly identifies anomalies instead of profiling normal data points • Can be scaled up to handle large, high-dimensional data	• Training time can be very long and computationally expensive • May produce inconsistent scores for particular points
Robust Random Cut Forest	• Analyzes real-time streaming data • Does not need to be reprogrammed for each learning	• Is time-consuming to construct trees and forests • Requires more computational resources

TABLE 2.1: Popular Anomaly Detection Algorithms

2.3 Anomaly Detection using Machine Learning

2.3.1 Artificial Intelligence

The notion of intelligence can be defined in many different ways. In computer science, it usually refers to an ability to take the right decisions, according to some criterion. To make better decisions requires adequate knowledge in a form that is operational. For instance, a machine should be able to interpret sensory data and then use that kind of information to make rational decisions.

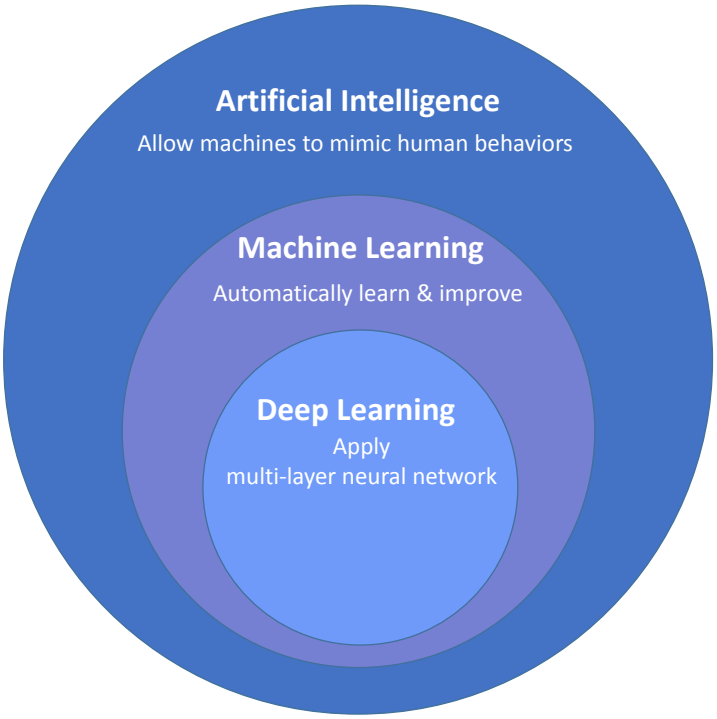


FIGURE 2.2: Relationship between Artificial Intelligence, Machine Learning and Deep Learning

Computers already possess some intelligence thanks to all the programs that humans have written over the past decades. These programs allow computers to do some things that are considered useful. In other words, it means that computers can make the "right" decisions given a series of the inputs. However, in contradiction of natural intelligence that presented by humans or animals, there are still a lot of tasks which animals and humans are able to do rather easily but still remain out of reach of computers. Many of these tasks fall into the category of artificial intelligence, to which include many perception and control tasks. Colloquially, the term "artificial intelligence" is applied when a machine mimics "cognitive" functions that humans associate with human mind, such as "learning" and "problem solving" (Stuart J. Russell 2009). Using data, examples and rules to build operational knowledge is what learning is about.

2.3.2 Machine Learning

Originated from pattern recognition, machine learning (ML) is a method of data analysis that gives computers the ability to learn without being explicitly programmed (Samuel 1959). Machine learning explores the study and construction of algorithms that can learn from data, identify patterns and make decisions (Ron Kohavi 1998). Instead of executing strictly static programmed instructions, ML algorithms build models from training sets of input and then make predictions, classifications or decisions expressed as outputs based on the developed models.

Even though a lot of machine learning algorithms have emerged, the ability of computers to automatically apply sophisticated mathematical computation to large volume of data is quite recent. Here are some real-world examples of machine learning applications:

- **Voice Recognition** Voice recognition is the ability of a system to understand voice dictation and carry out spoken commands. For example, Microsoft Cortana, Amazon Alexa and Apple Siri apply both machine learning and deep learning to imitate human interaction. As a consequence, they have the ability to gradually learn and understand human languages and even give the appropriate answers to questions.
- **Facial Recognition** Facebook's DeepFace can recognize the differences on human faces with accuracy very close to a human being. Making use of some machine learning technologies and algorithms, DeepFace can identify familiar faces from a user's contact list.
- **Semantic Recognition** In 2015, Google introduced RankBrain which utilizes an intuitive neural network. RankBrain employs algorithms which can decipher the semantic content of search engine queries and then offer tailored information.

2.3.3 Machine Learning Methods

Machine learning algorithms can be categorized as supervised, unsupervised, semi-supervised and reinforcement learning (Wilcox, Woon, and Aung 2013).

- **Supervised Learning**

  Supervised learning requires labelled data for training. In the training data set, each instance consists of an input object and a desired output in order to provide a learning basis. During the training stage, the supervised learning algorithm analyzes the training set to generate a model. After the training is done, the generated model can be applied to a new data set to make decisions or predictions.

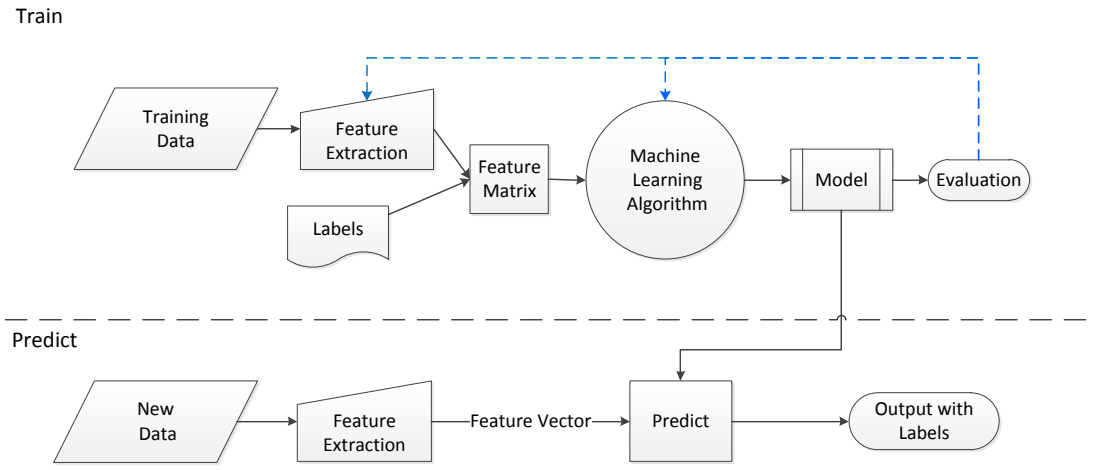


FIGURE 2.3: Supervised Learning

- **Unsupervised Learning**

  In contrast to supervised learning, unsupervised learning does not require either desired classified or labeled test data. Unsupervised learning algorithm is able to infer a function to describe hidden data structures from unclassified or uncategorized test data without guidance. It can be used for more complicated processing tasks than supervised learning algorithms. One common example is clustering for exploratory analysis, which is used to find hidden patterns in data set or grouping a set of instances.

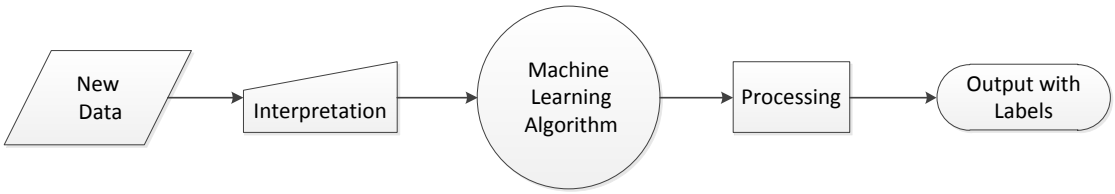


FIGURE 2.4: Unsupervised Learning

- **Semi-supervised Learning**

  This lies between unsupervised and supervised learning. It is similar to supervised learning but some of the training data are not labeled with the desired output. Semi-supervised learning performs well when there is only a very small amount of labeled data in a training set. Many studies suggest that in comparison with supervised learning which uses only labeled data, predictions with semi-supervised learning will be more accurate by considering the unlabeled input data. However, this is only accurate if some assumptions hold (Olivier Chapelle 2006), such as instances are more likely to share a label if they are close

to each other or instances in the same cluster are more likely to share a label when the data tend to form discrete clusters (Jie Wang and Xu 2019).

- **Reinforcement Learning**

  This is used to resolve the problem that to maximize outcome of cumulative reward in the long term, what kind of behaviors or actions should software agents take in an environment (Pack Kaelbling, Littman, and P Moore 1996). The environment takes the current state and action as an input and returns the next state and reward; the agent takes actions and receives the new state and reward from the environment and then figures out the best actions to take.

  Reinforcement learning is iterative and in most of the applications, the process begins without any clue about which state-action and reward will result.

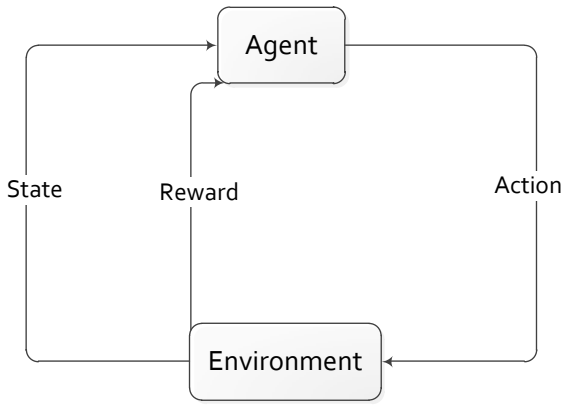


FIGURE 2.5: Framework of Reinforcement Learning

Lazy Learning

Lazy Learning (also known as memory-based learning, instance-based learning and locally weighted learning), defers processing of the examples until an explicit request for information is received (Atkeson, Moore, and Schaal 1997). That means in contrast to eager learning which generalizes the training data into a model before receiving any queries, lazy learning does not have a training phase. During the generalization, the whole data set is searched for the most relevant instances to answer the request according to some criteria.

One advantage of lazy learning is that it is particularly suitable for examples that are not all available from the beginning but are collected constantly (i.e., website visits or wireless signals). Another benefit is that lazy learning does not suffer from data interference (Atkeson, Moore, and Schaal 1997). Catastrophic data interference is a tendency of an artificial neural network to completely and abruptly forget previously learned information upon learning new information (McCloskey and Cohen 1989). It is caused by the negative interference between original and new training data. Algorithms forget the old data as that data is not included in the new

training data set. Lazy learning avoids the potential interference as it retains all the original data.

2.3.4 Anomaly

Anomalies, also known as outliers, novelties or noise, are a subset of instances that are substantially different from the rest of the data set. An outlier may happen because of measurement variability or might occur because of some experimental errors that are often excluded from the data set (Grubbs 1969).

An anomaly may cause serious problems in data analysis and lead to improper decisions. Anomalies may be introduced into the data set due to various reasons. For instance, hardware which is used for taking measurements might experience a transient malfunction. Sometimes, errors occur during data transmission. Anomalies can also happen because of system behavior changes, instrument errors, human impacts or fraudulent behaviors.

Anomaly Types

Generally speaking, anomalies can be categorized into the following three types (Arindam Banerjee and Kumar 2009):

Point Anomalies

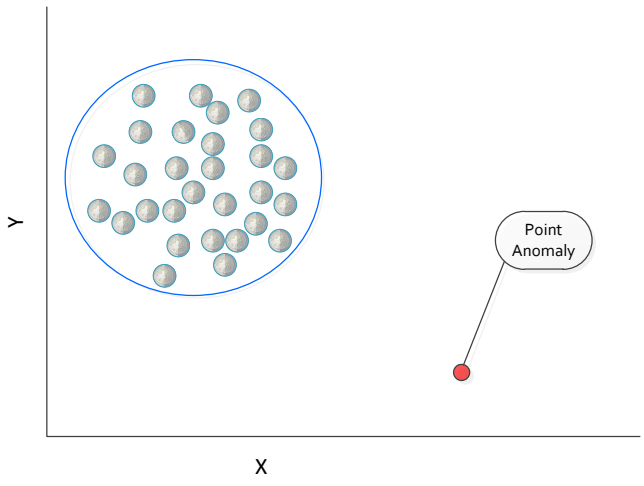


FIGURE 2.6: Point Anomaly

A point anomaly is an individual data point that is far away from others and therefore can be considered anomalous. Point anomalies are also called global anomalies and are the simplest anomaly type. Point anomaly detection is the main focus of most research.

Figure 2.6 illustrates a point anomaly. The red point lies outside the boundary of the normal data cluster (the grey points), and hence is a point anomaly as it is different from other normal data points.

Consider credit card fraud detection as a real-life example. One transaction will be classified as a point anomaly if its amount is high compared to the normal range of expenditures.

Contextual Anomalies

If a data instance is anomalous in a specific context, but not otherwise, then it is termed as a contextual anomaly (also referred to as a conditional anomaly) (Song, Wu, Jermaine, and Ranka 2007). Contextual anomalies are what this research is trying to detect in water quality data.

Contextual anomalies are very common in time series data. For example, spending $150 on food every day may be odd but might be quite normal during the holiday season. Figure 2.7 shows an example for a temperature time series. In this example, a temperature of 5 Celsius might be an anomaly during that period of time, but the same value would be considered as normal in winter.

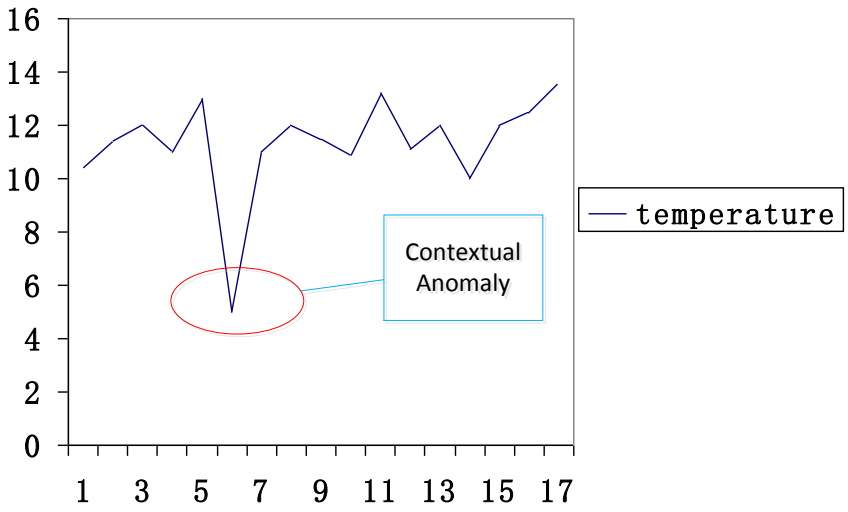


FIGURE 2.7: Contextual Anomaly

Collective Anomalies

Collective anomalies are a collection of data instances which are considered as anomalous when compared to the entire data set. Under some circumstances, the individual data instance of collective anomalies may not be considered as anomaly.

Figure 2.8 shows a group of star points that are considered collective anomaly to the collection of grey points, even though the stars which are very close to the grey points might be normal data individually.

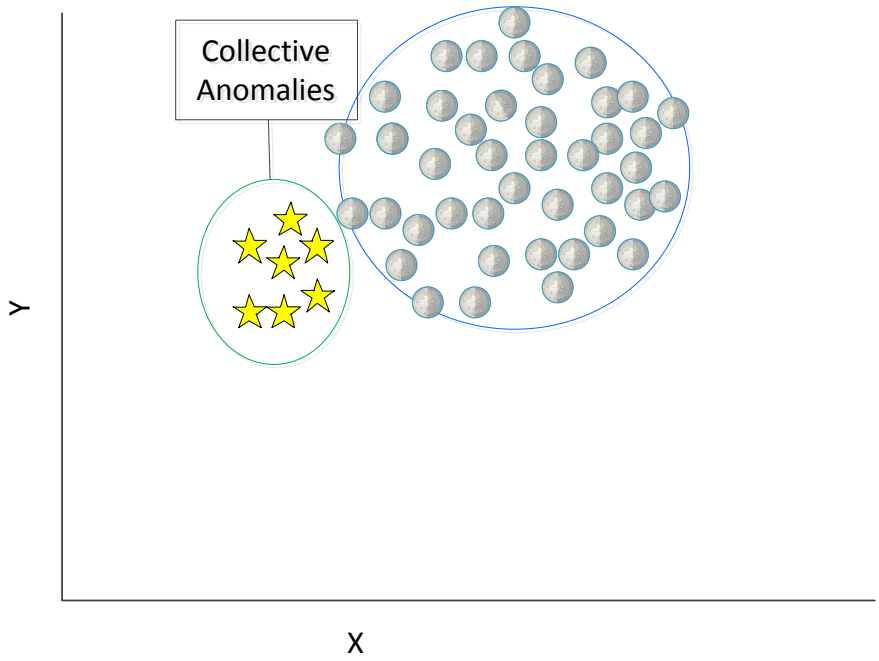


FIGURE 2.8: Collective Anomalies

2.3.5 Local Outlier Factor

k-Nearest Neighbors

The k-nearest neighbors algorithm (k-NN) is a non-parametric and lazy learning method which has been used for classification and regression (Altman 1992). Non-parametric methods are mathematical procedures that do not require the population to meet certain assumptions, or parameters. The nearest neighbor methods are used to find a certain number of instances in the training data set that are closest to new data using distance measurements, and then make a prediction or classification of the label of the new data based on these instances. This has been studied for quite a while:

- Instance-Based Learning (also known as Memory-based learning): The raw training data is stored in memory and used to make predictions.
- Lazy Learning: Lazy learning means that it does not explicitly generate a model at the beginning as discussed in Section 2.3.3. Instead, it loads the whole training data set into memory for the classification phase. Thus, there is no learning phase for model generation and all the tasks execute at the time when a prediction is requested.
- Non-Parametric: It means that k-NN does not make any assumption about the functional forms of the issue to be resolved.
- k-NN can be used for both regression and classification (Altman 1992).

The k-NN algorithm is a very robust and versatile classifier and is one of the simplest machine learning algorithms. Even though it is pretty simple, k-NN can outperform very powerful classifiers and has been successful in many classification and regression problems. As such, it has been used in various applications such as data compression, economic forecasting, and genetics.

Figure 2.9 shows the classification of a point based on its closest three neighbors. As it is a lazy learning method, k-NN makes predictions using the training data set directly instead of constructing a general internal model in advance. For prediction, the new instance is predicted by querying the entire data set for the k nearest instances at first and then summarizing the output of each instance. And for regression, the result might be the average of the values of its k nearest neighbors. For classification, the result is the class which defined by the plurality vote of its k nearest neighbors based on the calculation of distance.

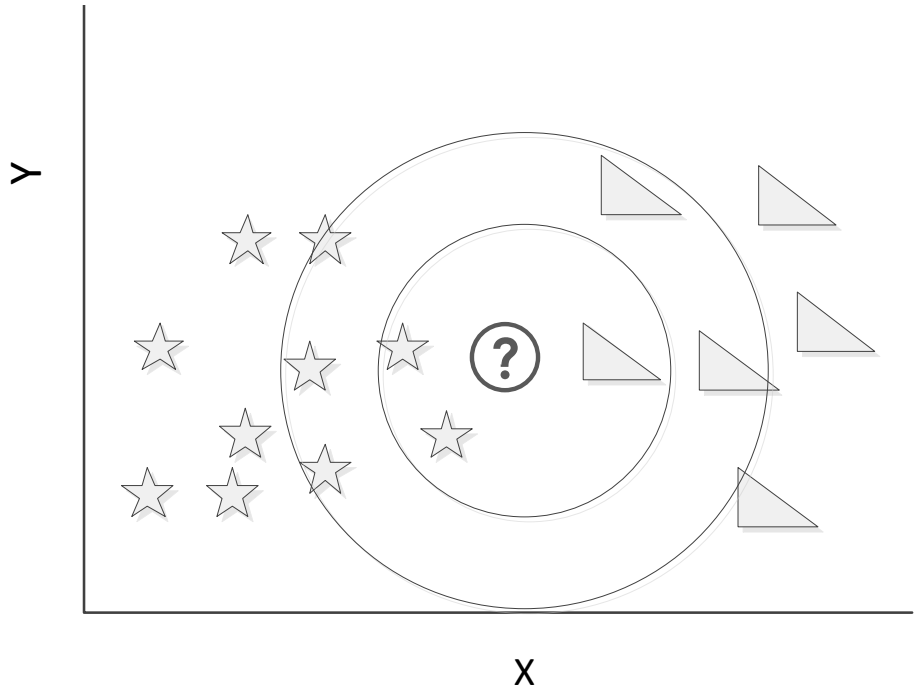


FIGURE 2.9: k-Nearest Neighbors

The value for k is an integer which is specified by the user. It is highly data-dependent, and it is better to try as many different values as possible during algorithm tuning to find the suitable k. However, one of the disadvantages is that while the size of the training data set grows, the computational complexity of k-NN rises as well because this algorithm must store all the data in memory to compute the result. Alternatively, for large training sets, k-NN could be stochastic by dividing the training data set into small portions in order to calculate the k-most closest instances.

In general, the distance can be any metric measure to determine how many of the k instances are closest to a new instance in the training data set. Euclidean distance is the most common

choice for real-valued input variables. Generally speaking, for an n-dimensional space, the Euclidean distance between points p and q can be defined as:

$$distance(p,q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \cdots + (p_{n-1} - q_{n-1})^2 + (p_n - q_n)^2} \tag{2.1}$$

There are several other popular distance measures:

- Hamming Distance: This concept was introduced by Richard Hamming in his paper (Hamming 1950). It is used to calculate the distance between binary vectors, but it can only be calculated for lines that are the same length. Calculation of the Hamming distance is not complicated and works by simply adding up the number of spots where the values of the lines are different. The Hamming Distance is defined as:

  Given two binary vectors v_1 and v_2, the Hamming Distances between v_1 and v_2 is the number of digits where v_1 and v_2 defer.

  For instance, the distance between the two binary numbers 10100101 and 10101010 is four as only the last four digital are different.

- Manhattan Distance: This is also known as L^1 distance or City Block distance. The L^1 metric was first used in regression analysis in 1757 by Roger Joseph Boscovich (Stigler 1986).The Manhattan distance between two point p and q in n-dimensions is defined as Equation 2.2. This is the distance between real vectors which calculated by sum of their horizontal and vertical distances. It works very well when the instances are not similar in type.

$$distance(\mathbf{p},\mathbf{q}) = \sum_{i=1}^{n} \mid p_i - q_i \mid \tag{2.2}$$

- Minkowski Distance: This is named after the German mathematician Hermann Minkowski. This can be considered as a generalization of both the Euclidean distance and the Manhattan distance (Merigo and Casanovas 2012). The Minkowski Distance of order λ between two points p and q in n-dimensions is defined as Equation 2.3. Actually, it is equivalent to the Manhattan Distance when $\lambda = 1$ and becomes the Euclidean Distance when $\lambda = 2$.

$$distance(\mathbf{p},\mathbf{q}) = \left(\sum_{i=1}^{n} |p_i - q_i|^{\lambda}\right)^{1/\lambda} \quad \text{where} \quad \lambda \in \mathbb{R} \tag{2.3}$$

Other distance measures can also be used like Tanimoto, Jaccard and Mahalanobis. Alternatively, a user-defined function of the distance can be used for calculation as well.

The distance to the k^{th} nearest neighbor can be used as an outlier score in anomaly detection as it can be considered as estimation of a local density. The larger the distance to the k-NN, the lower the local density, and the more likely the query point is an outlier (Angiulli 2005). Although quite simple, this outlier model, along with another classic data mining method, local

outlier factor, works quite well in comparison to more recent and more complex approaches, according to a large scale experimental analysis (Garcia, Derrac, Cano, and Herrera 2012).

Local Outlier Factor

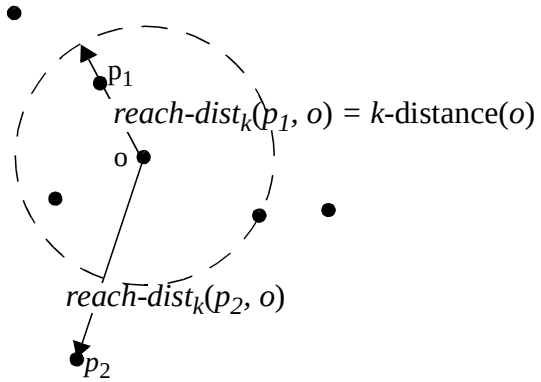
The local outlier factor (LOF) is an unsupervised outlier detection algorithm that detects the outliers by comparing the local density of the data instance with its neighbors (Breunig, Kriegel, Ng, and Sander 2000). It was the first algorithm based on local density and k-neighborhood. The anomaly score of each sample in the training data set is called local outlier factor and indicates its outlier-ness degree. LOF has been well studied and developed because it has the capability to detect anomalies that were unperceived by the global methods.

The LOF of an instance is based on the number of nearest neighbors which are used to determine its local neighborhood. The following definitions are used for the LOF to accomplish the whole process. The k-distance of instance p, denoted as k-distance(p), is defined as the distance $d(p, o)$ between p and an object $o \in D$ so that for at least k instances $o' \in D\backslash\{p\}$ it holds that $d(p, o') \leq d(p, o)$, and for at most $k-1$ instances $o' \in D\backslash\{p\}$ it holds that $d(p, o') < d(p, o)$ (Breunig, Kriegel, Ng, and Sander 2000).

The k-distance neighborhood of instance p is a subset that contains the instances whose distances from it are not greater than the k-distance.

The definition of reachability distance of instance p in regard to instance o is:

$$reach\text{-}dist_k(p, o) = max\{k\text{-}distance(o), d(p, o)\} \tag{2.4}$$



Source: (Breunig, Kriegel, Ng, and Sander 2000)
FIGURE 2.10: Examples of Reachability Distance for $k = 4$

Figure 2.10 shows examples of reachability distance for $k = 4$. Generally speaking, the reachability distance between these two instances is their actual distance if they are far away from each other(like o and p_2 in the above figure); but, the reachability distance is *k-distance*

of o if they are close enough (like o and p_1 in the above figure). As a consequence, the statistical fluctuations of $d(p, o)$ for all of the $p's$ close to o can be significantly reduced.

The parameter k controls the strength of this smoothing effect so that the higher the value of k, the more similar the reachability distances are for instances within the same neighborhood (Breunig, Kriegel, Ng, and Sander 2000).

For object $o \in N_{MinPts}(p)$, the definition of the local reachability density of point p is:

$$lrd_{MinPts}(p) = 1 \Big/ \left(\frac{\sum_{o \in N_{MinPts}(p)} reach\text{-}dist_{MinPts}(p, o)}{|N_{MinPts}(p)|} \right) \tag{2.5}$$

where

- $MinPts$ specifys a minimum number of objects
- $reach\text{-}dist_{MinPts}(p, o)$ represents the reachability distance of object p with respect to object o

For object $o \in N_{MinPts}(p)$, the definition of (local) outlier factor of p is:

$$LOF_{MinPts}(p) = 1/(\frac{\sum_{o \in N_{MinPts}(p)} \frac{lrd_{MinPts}(o)}{lrd_{MinPts}(p)}}{|N_{MinPts}(p)|}) \tag{2.6}$$

where

- $lrd_{MinPts}(p)$ is the local reachability density of p
- $lrd_{MinPts}(o)$ represents the local reachability density of p's $MinPts$-nearest neighbors

There are five steps to calculate the LOF for a data set:

1. Calculate all the distances between each two instances
2. Calculate all the distances between p and its k^{th} nearest neighbors
3. Calculate all the k nearest neighbors of p
4. Calculate the Local Reachability Density of p
5. Calculate all the LOFs of p

LOF vibrates with different size of neighborhood. Thus, a range for the size of the neighborhood should be defined in order to improve the results. Therefore, the final score is the maximum LOF score over that range. Instances with LOF scores of approximately equal to 1 are considered as normal data, whereas instances with scores much greater than 1 would be anomalies. Because local density would be similar to its neighbors if the instance lies within a cluster. Then a score of this instance equals to 1. The problem is that there is no clear rule for the threshold for which a point can definitely considered as an anomaly. For a clean data set without large fluctuations, an instance with LOF score of around 2 would be an anomaly. Whereas an instance with LOF of 20 still is an inlier for a dirty data set with a lot of large fluctuations.

Choosing optimal k is essential for detection performance and it is impossible to estimate the optimal k based on the training data. If k is too small, unwanted statistical fluctuations will affect the result, but if k is too large, the error will go up again due to under-fitting. The author of LOF suggested in his paper (Breunig, Kriegel, Ng, and Sander 2000) that k in the range of 10 to 50 generally works well.

The following is a simple implementation of Local Outlier Factor written in Python.

```
import pickle
import pandas as pd
import numpy as np
from itertools import combinations
from math import inf
from math import sqrt, inf


def read_data(file_name):
    '''
    Retrieves data from text file
    and stores as data frame using pandas
    '''
    data_frame = pd.read_table(file_name, header=None, delim_whitespace=True)
    data_frame = data_frame.iloc[:, :-1]
    data_frame = (data_frame - data_frame.min()) / (data_frame.max() - data_frame
    .min())
    return data_frame.values


def get_euclidean_distance(p, q):
    '''
    Calculates the euclidean distance
    between two instances of the data
    '''
    return sqrt((p['x'] - p['x']) ** 2 + abs(q['y'] - q['y']) ** 2)


def create_distance_matrix(data, N):
    '''
    Computes the distance matrix and then
    writes to to a pickle file in order to
    save time  on future runs
    '''
    distance_matrix = np.zeros((N, N))
    i = 0
    for a in data:
        j = 0
        for bin data:
            distance_matrix[i][j] = get_euclidean_distance(a, b)
        j += 1
    i += 1
    f = open('distancematrix', 'wb')
    pickle.dump(distance_matrix, f)
    f.close()
    return distance_matrix


def get_local_reachability_density(N, distance_matrix, k, data):
```

```
    '''
    Finds local reachability density for each sample by the following steps:
    1. Calculate all the distances for each instances

    2. Calculate the number of points that fall within the k-distance
    neighbourhood
    3. Calculate reachability distances
    4. Calculate all the Local Reachability Density of p

    '''
    k_dist = np.zeros(N)
    k_neighbours = {}
    num_of_neighbours = 0
    lrd = np.zeros(N)

    for i in range(N):
        distance_point = distance_matrix[i]
        knn = np.partition(distance_point, k-1)
        k_dist[i] = knn[k-1]
        sort_index = np.argsort(distance_point)

        j = 0
        temp = []
        for dist in distance_point:
            if dist <= k_dist[i]:
                temp.append(sort_index[j])
                num_of_neighbours += 1
            j += 1
        k_neighbours[i] = temp

    reachability_distance = get_reachability_distances(N, data, k_dist,
    distance_matrix)

    for i in range(N):
        sum_of_reachability_distances = 0
        for value in k_neighbours[i]:
            sum_of_reachability_distances += reachability_distance[int(value)][i]
        if sum_of_reachability_distances == 0:
            lrd[i] = inf
        lrd[i] = len(k_neighbours[i])/sum_of_reachability_distances

    return lrd


def get_reachability_distances(N, data, k_dist, distance_matrix):
    '''
    Calculates the reachability distance
    between each two instances
    '''
    reachability_distance = np.zeros((N, N))
    i = 0
    for _ in data:
        j = 0
        for _ in data:
            reachability_distance[i][j] = max(k_dist[i], distance_matrix[i][j])
            j += 1
        i += 1
```

```
104     return reachability_distance
105
106
107 def main(file_name, k):
108     '''
109     Gets distance matrix, the LRD, and the first O
110     points after sorting of LRD
111
112     Parameters
113     ----------
114     file_name :  The name of the data file
115     k : to get kNN, kdist, and then LOF
116     '''
117     data = read_Data(file_name)
118     total_num = len(data)
119     distance_matrix = create_distance_matrix(data, N)
120     lrd = get_local_reachability_density(total_num , distance_matrix, k, data)
121     sorted_outlier_factor_indexes = np.argsort(lrd)
122     outliers = sorted_outlier_factor_indexes[-O:]
123
124
125 if __name__ == '__main__':
126     main()
```

2.3.6 Example for Anomaly Detection with LOF

The following Jupyter Notebook shows how to use LOF for anomaly detection. Table 2.2 shows the data set used for this example.

Timestamp	Value
2018/1/1 0:00	8.2
2018/1/1 0:01	8.1
2018/1/1 0:02	8
2018/1/1 0:03	8.1
2018/1/1 0:04	8.2
2018/1/1 0:05	8.2
2018/1/1 0:06	8.1
2018/1/1 0:07	8.1
2018/1/1 0:08	8
2018/1/1 0:09	8.2
2018/1/1 0:10	8.1
2018/1/1 0:11	8.2
2018/1/1 0:12	8
2018/1/1 0:13	8.2
2018/1/1 0:14	8.1
2018/1/1 0:15	1
2018/1/1 0:16	8
2018/1/1 0:17	8.2
2018/1/1 0:18	8
2018/1/1 0:19	8.2

TABLE 2.2: Test Data

In the above data set, the value "1" at "2018/1/1 0:15" is the anomaly. The following example shows that LOF successfully detects this anomaly.

Jupyter Notebook

Run the reusable Python code for library importing and auxiliary functions. Details of this file are in Section 3.2.

```
In [1]: %run anomaly_detection.py
```

Get temperature test data from "TestData.csv" file.

```
In [2]: test_data = pd.read_csv("TestData.csv")
```

Display the test data.

```
In [3]: print(test_data)
```

```
             Timestamp  Value
0   2018-01-01 00:00:00    8.2
1   2018-01-01 00:01:00    8.1
2   2018-01-01 00:02:00    8.0
3   2018-01-01 00:03:00    8.1
4   2018-01-01 00:04:00    8.2
```

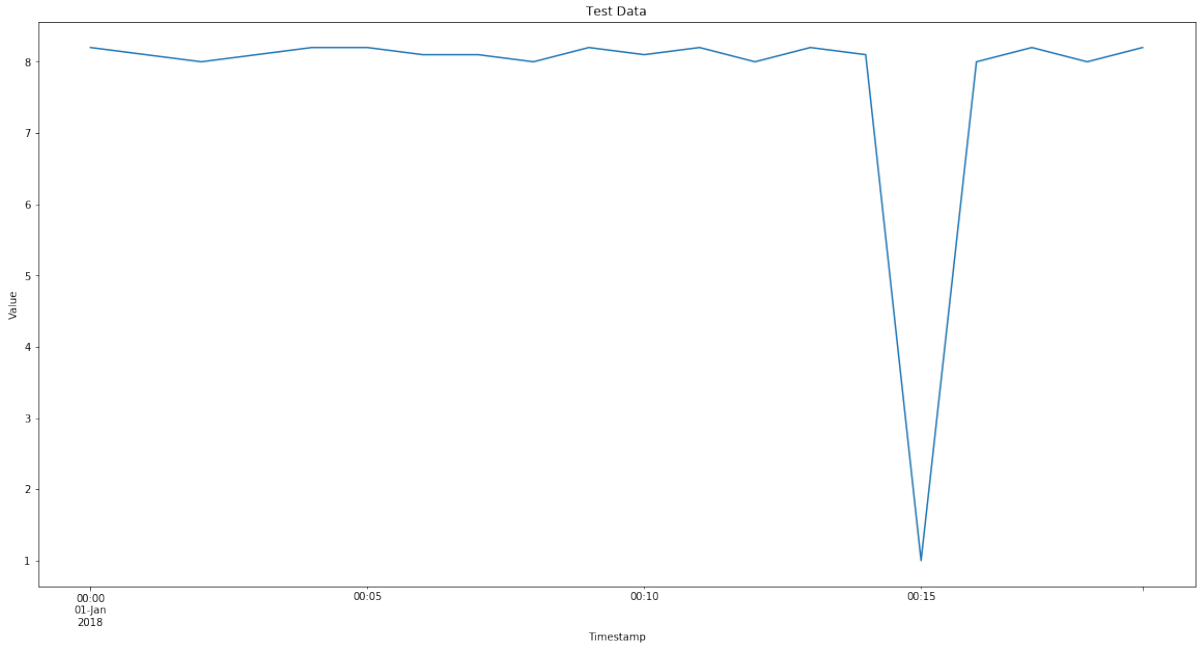
```
5   2018-01-01 00:05:00    8.2
6   2018-01-01 00:06:00    8.1
7   2018-01-01 00:07:00    8.1
8   2018-01-01 00:08:00    8.0
9   2018-01-01 00:09:00    8.2
10  2018-01-01 00:10:00    8.1
11  2018-01-01 00:11:00    8.2
12  2018-01-01 00:12:00    8.0
13  2018-01-01 00:13:00    8.2
14  2018-01-01 00:14:00    8.1
15  2018-01-01 00:15:00    1.0
16  2018-01-01 00:16:00    8.0
17  2018-01-01 00:17:00    8.2
18  2018-01-01 00:18:00    8.0
19  2018-01-01 00:19:00    8.2
```

Function process_data is a function to parse "Timestamp" column as date time and set it as index.

```
In [4]: processed_data = process_data(test_data)
```

Plot the test data.

```
In [5]: f1 = plt.figure(figsize=(20,10))
        test_data['Value'].plot()
        plt.xlabel("Timestamp")
        plt.ylabel("Value")
        plt.title("Test Data")
        plt.show()
```



Reshape "Value" column test data as training data.

```
In [6]: training_data = prepare_training_dataset(processed_data)
```

Initialize an instance for outlier detection with n_neighbors(k) = 10

```
In [7]: clf = LocalOutlierFactor(n_neighbors=10)
```

Make prediction of training data set.

```
In [8]: prediction = lof_prediction(clf,traing_data, processed_data)
```

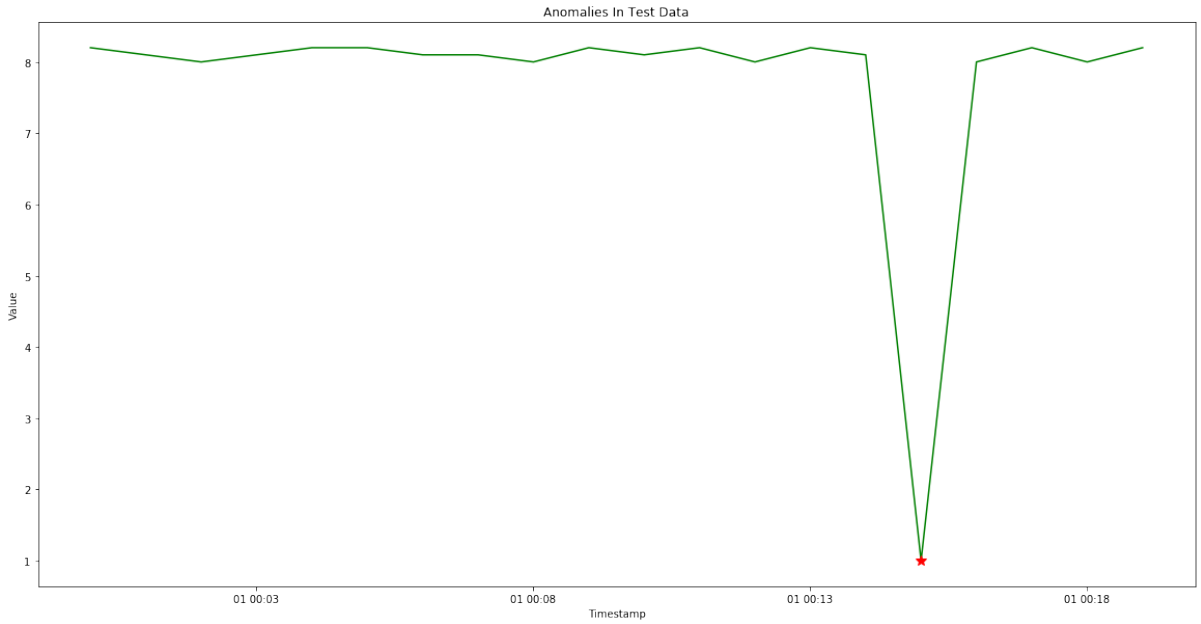
Display detection result.

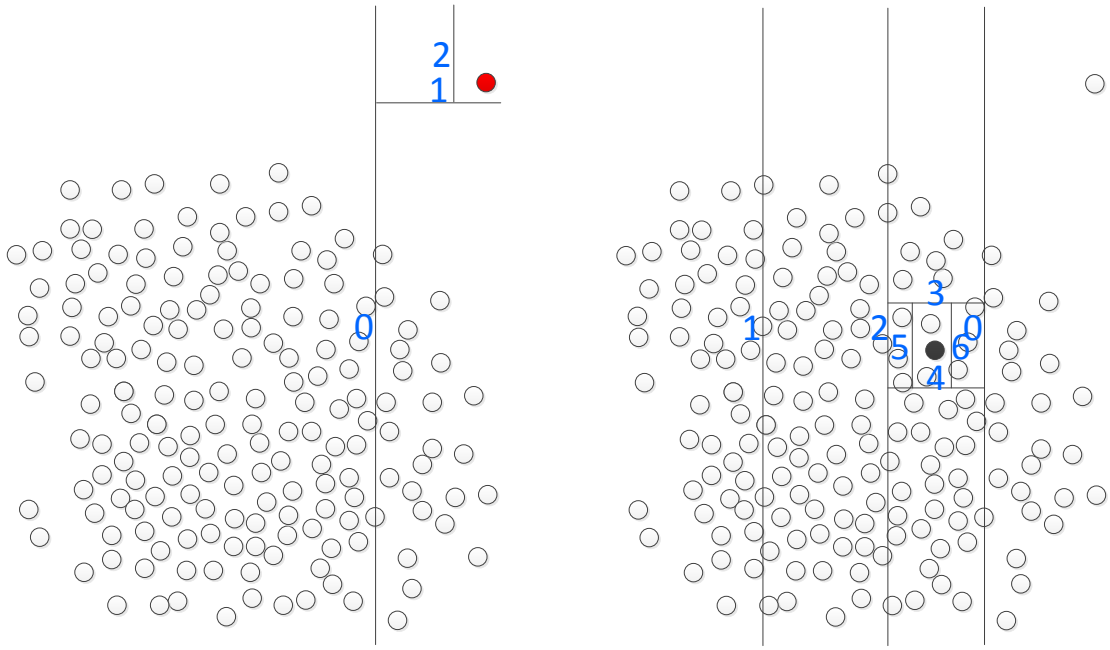
```
In [9]: print(prediction)
```

```
                     Value  isinlier
Timestamp
2018-01-01 00:15:00    1.0    -63.55
```

Plot the result.

```
In [10]: plot_lof_result(processed_data,prediction,'Anomalies In Test Data')
```





(A) Anomaly (B) Normal Point

FIGURE 2.11: Identifying Normal and Abnormal Instances

2.3.7 Isolation Forest

Isolation Forest (IF) was proposed by (Liu, Ting, and Zhou 2012) to identify anomalies in a data set purely based on isolation without relying on any distance or density measure. Isolation means separating an instance from the rest of the instances. IF was used by (Ding and Fei 2013) to detect outliers and change points from non-stationary time series data.

IF is an unsupervised algorithm for detecting outliers within a data set. It utilizes two quantitative properties for anomalies:

- There are only a few anomalies in the data set.
- Anomalies have attributes that are distinct from normal data.

A binary tree structure called an isolation Tree (iTree) is used to isolate instances. In an iTree, each node has either zero or two daughter nodes. Let $X = x_1, \cdots, x_n$ be the given data set of a d-variate distribution. A sample of instances $X_0 \cdot X$ is used to build an isolation tree (iTree). X_0 is recursively divided by a randomly selected attribute q and a split value p, until either the node has only one instance or all data at the node have the same value.

In an iTree, the anomalies are isolated and closer to the root of the tree, while the normal instances are isolated at the deep end of the tree. Figure 2.11 denotes the idea of identifying a normal and abnormal instances using iTree. It shows that the abnormal point (red point in diagram A) can be directly isolated, while the normal point (black point in diagram B) requires multiple random partitions to be isolated.

Each instance is given an anomaly score for decision making. The anomaly score s of an instance x is defined as:

$$s(x,n) = 2^{-\frac{E(h(x))}{c(n)}} \tag{2.7}$$

where

- $h(x)$ is the path length of observation x.
- $c(n)$ is the average path length of unsuccessful search in a Binary Search Tree.
- n is the number of external nodes.

The score range is [0,1]. The following decision can be made based on the scores:

- Score close to 1 indicates anomaly.
- Score much smaller than 0.5 indicates a normal instance.
- There is no distinct anomaly in the entire data set if all scores are close to 0.5.

Extended Isolation Forest (EIF) was proposed by (Hariri, Carrasco Kind, and Brunner 2018) to improve the consistency and reliability of the anomaly score of a given instance. Unlike IF which selects a random attribute and random value, EIF randomly selects slopes for the branch cut and randomly chooses intercepts from available values within the training data set.

2.3.8 Robust Random Cut Forest

Robust Random Cut Forest (RRCF) was proposed by (Guha, Mishra, Roy, and Schrijvers 2016). It is an unsupervised algorithm for anomaly detection on streaming data. Current records in the stream are used to develop the machine learning model. RRCF does not use older records nor does it use statistics from previous executions.

The following is the procedure of anomaly detection using RRCF:

1. RRCF takes a bunch of random instances (Random).
2. Then, it cuts them into the same number of instances and creates trees (Cut).
3. Finally, it determines whether a particular instance is an anomaly by considering all of the trees together (Forest).

A Robust Random Cut Tree (RRCT) on point set S is generated as follows (Guha, Mishra, Roy, and Schrijvers 2016):

1. Choose a random dimension proportional to $\frac{\ell_i}{\sum_j \ell_j}$ where $\ell_i = max_{x \in S} x_i - min_{x \in S} x_i$
2. Choose $X_i \sim Uniform[min_{x \in S} xi, max_{x \in S} x_i]$
3. Let $S1 = \{x | x \in S, x_i \leqslant X_i\}$ and $S2 = S \backslash S1$ and recurse on $S1$ and $S2$

Figure 2.12 shows the idea how RRCF cut instance into pieces recursively. The cutting stops when each point is isolated.

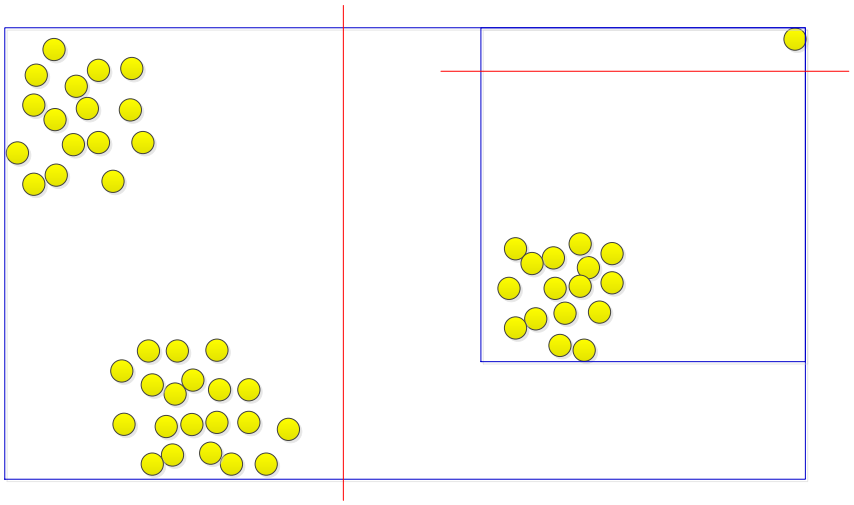


FIGURE 2.12: Random Cut Tree

Robust Random Cut Trees can be dynamically maintained using Insertion and Deletion operations when RRCF is used to detect anomalies on stream data.

- Deletion: If T were drawn from the distribution $RRCF(S)$ then Algorithm 1 produces a tree T' which is drawn at random from the probability distribution $RRCF(S - \{p\})$
- Insertion: Given T drawn from distribution $RRCF(S)$ and $p \in S$ produce a T' drawn from $RRCF(S \cup p)$. The algorithm is provided in Algorithm 2.

Algorithm 1 ForgetPoint

1: Find the node v in the tree where p is isolated in T.
2: Let u be the sibling of v . Delete the parent of v (and of u) and replace that parent with u (i.e., short circuit the path from u to the root).
3: Update all bounding boxes starting from u's (new) parent upwards –this state is not necessary for deletions, but is useful for insertions
4: Return the modified tree T.

Source: (Guha, Mishra, Roy, and Schrijvers 2016)

Algorithm 2 Algorithm InsertPoint

1: Given a set of points S' and a tree $T(S')$. Insert new point p and produce tree $T'(S' \cup \{p\})$.

2: If $S' = \phi$ then return a node containing the single node p.
3: Otherwise S' has a bounding box $B(S') = [x_1^\ell, x_1^h] \times [x_2^\ell, x_2^h] \times \ldots [x_d^\ell, x_d^h]$. Let $x_i^\ell \leq x_i^h$ for all i.
4: For all i let $\hat{x}_i^\ell = min\{p_i, x_i^\ell\}$ and $x\hat{}h_i = max\{x_i^h, p_i\}$.
5: Choose a random number$r \in [0, \sum_i (\hat{x}_i^h - \hat{x}_i^\ell)]$.
6: This r corresponds to a specific choice of a cut in the construction of RRCF $(S' \cup \{p\})$. For instance, compute arg $min\{j | \sum_{i=1}^{j} (\hat{x}_i^h - \hat{x}_i^\ell) \geq r\}$ and the cut corresponds to choosing $\hat{x}_j^\ell + \sum_{i=1}^{j} (\hat{x}_i^h - \hat{x}_i^\ell) - r$ in dimension j.
7: If this cut separates S' and p (i.e., is not in the interval $[x_j^\ell, x_j^h]$) then use this as the first cut for $T'(S' \cup \{p\})$. Create a node –one side of the cut is p and the other side of the node is the tree $T(S')$.
8: If this cut does not separate S' and p then throw away the cut! Choose the exact same dimension as $T(S')$ in $T'(S' \cup \{p\})$ and the exact same value of the cut chosen by $T(S')$ and perform the split. The point p goes to one of the sides, say with subset S''. Repeat this procedure with a smaller bounding box $B(S'')$ of S''. For the other side, use the same subtree as in $T(S')$.
9: In either case update the bounding box of T'.

Source: (Guha, Mishra, Roy, and Schrijvers 2016)

RRCF assigns an anomaly score to each instance. The anomaly score of a point is defined by its collusive displacement, which measures the change in model complexity incurred by inserting or deleting a given point p (Guha, Mishra, Roy, and Schrijvers 2016). Low scores indicate that the instances are considered as normal. High scores indicate the presence of anomalies in the data set.

2.4 Statistical Techniques

2.4.1 Seasonal Hybrid Extreme Studentized Deviate

Seasonal Hybrid Extreme Studentized Deviate (S-H-ESD) is an algorithm developed by Twitter in 2015 to detect when anomalies occurred in the corresponding time series. It has the ability to detect both point and contextual anomalies through applying time series decomposition. It is built upon the generalized Extreme Studentized Deviate (ESD) test which is used to detect one or more outliers in a univariate data set that follows an approximately normal distribution (Rosner 1983).

The generalized ESD is defined based on the assumption of the dataset:

- There is no outlier in the data set.
- There are at most r outliers in the data set.

For a data set D of size n, given $x \in D$, essentially the generalized ESD test runs r separate tests to compute the largest absolute deviation from the sample mean in units of the sample standard deviation:

$$G_i = \frac{max_i|x_i - \overline{x}|}{s} \tag{2.8}$$

where

- $i = 1, 2, \cdots, r$.
- $\overline{x}$ is the sample mean.
- s is sample standard deviation.

This results in the r test statistics $G_1, G_2, \cdots, G_r$.

In order to determine whether a value is anomalous or not, the test result is then compared with a critical value which is calculated using following equation:

$$\lambda_i = \frac{(n-i)t_{p,n-i-1}}{\sqrt{(n-i-1+t^2_{p,n-i-1})(n-i+1)}} \tag{2.9}$$

where

- $i = 1, 2, \cdots, r$.
- $\overline{x}$ is the sample mean.
- $p = 1 - \frac{\alpha}{2(n-i+1)}$ and α is the significance level that represents the probability of rejecting the null hypothesis when it is true in statistics.
- $t_{p,n-i-1}$ represents the p^{th} percentile point from the t distribution with d degrees of freedom

The value will be deleted from the data set if it is considered to be anomalous and the critical value will be recalculated using the remaining data. ESD repeats this process for r times, and the largest $r(G_r > \lambda_r)$ is the number of anomalies in the data set.

Seasonal-ESD (S-ESD) uses a modified STL (seasonal-trend decomposition procedure based on Loess (Cleveland, Cleveland, McRae, and Terpenning 1990)) decomposition to extract the residual component of the input time series and then applies ESD to detect anomalies (Hochenbaum, Vallis, and Kejariwal 2017). Thus S-ESD has the ability to detect both global and local anomalies. However, S-ESD does not work well when applied to data sets that have a high percentage of anomalies. Therefore, Twitter introduced Seasonal Hybrid ESD (S-H-ESD) that adopts the S-ESD algorithm and uses the robust statistical techniques and metrics as discussed before. As a consequence, S-H-ESD is capable of detecting tendency of a time series with a high percentage of anomalies.

Twitter AnomalyDetection

Twitter AnomalyDetection is an open-source R package that can be used to detect anomalies by means of statistics. The primary algorithm of AnomalyDetection is S-H-ESD. This package was designed to detect both point and contextual anomalies and is very robust and versatile if data presents seasonality and trends. This complementary package can be used in various

applications. For example, it has been used for detecting anomalies in system metrics after a new software release, for problems in econometric, financial engineering, political and social sciences (Twitter 2015).

Twitter's AnomalyDetection is easy to use, but it is an R library. In the experiments, a Python implementation of S-H-ESD (Marcnuth 2018) was used for comparison of the performance. This implementation rewrites Twitter's Anomaly Detection algorithms in Python, but provides the same functions.

The repository (Marcnuth 2018) is the main Python implementation of Twitter's Anomaly-Detection for time series data.

2.4.2 Example for S-H-ESD

The following Jupyter Notebook shows how to use S-H-ESD for anomaly detection. Test data normal range is [8, 8.2]. There are two abnormal records: [2018-02-02 02:02:00, 7.0] and [2018-02-02 20:02:00, 8.9].

Run the reusable Python code for library importing and auxiliary functions.

```
In [1]: %run anomaly_detection.py
```

Get test data from csv file.

```
In [2]: test_data = pd.read_csv('TestData-ESD.csv')
```

Display the test data.

```
In [3]: print(test_data)
```

```
             Timestamp  Value
0     2018-02-02 0:00    8.2
1     2018-02-02 0:01    8.1
2     2018-02-02 0:02    8.0
3     2018-02-02 0:03    8.1
4     2018-02-02 0:04    8.2
5     2018-02-02 0:05    8.2
6     2018-02-02 0:06    8.1
7     2018-02-02 0:07    8.1
8     2018-02-02 0:08    8.0
9     2018-02-02 0:09    8.2
10    2018-02-02 0:10    8.1
11    2018-02-02 0:11    8.2
12    2018-02-02 0:12    8.0
13    2018-02-02 0:13    8.2
14    2018-02-02 0:14    8.1
15    2018-02-02 0:15    8.0
16    2018-02-02 0:16    8.0
17    2018-02-02 0:17    8.2
18    2018-02-02 0:18    8.0
```

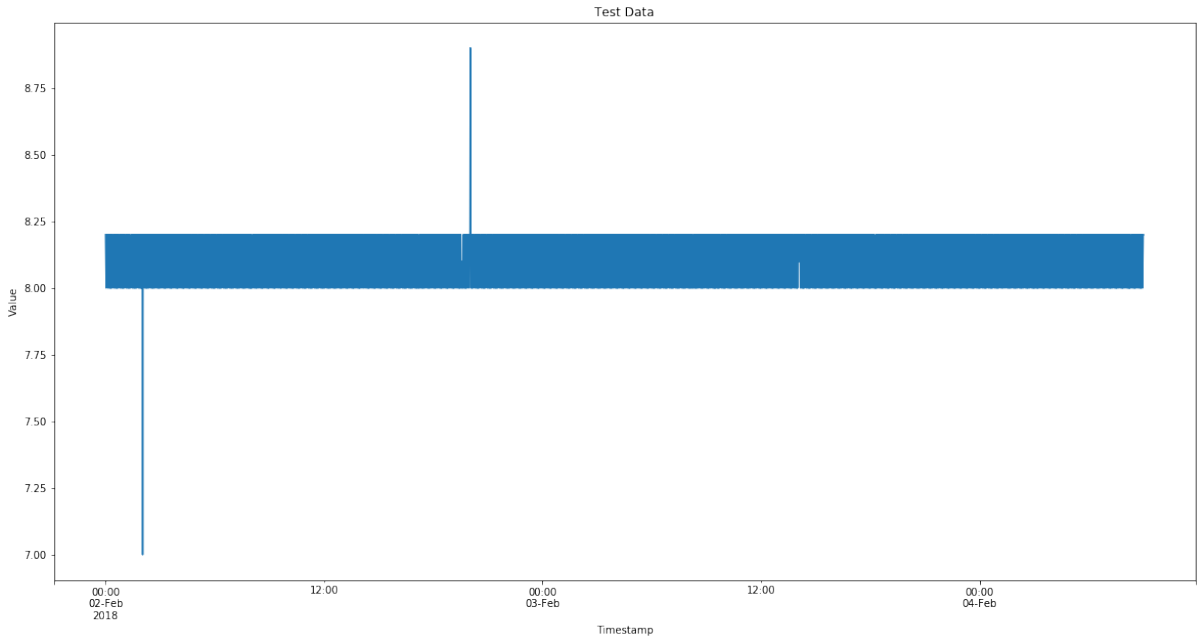
```
19    2018-02-02 0:19    8.2
20    2018-02-02 0:20    8.2
21    2018-02-02 0:21    8.1
22    2018-02-02 0:22    8.0
23    2018-02-02 0:23    8.1
24    2018-02-02 0:24    8.2
25    2018-02-02 0:25    8.2
26    2018-02-02 0:26    8.1
27    2018-02-02 0:27    8.1
28    2018-02-02 0:28    8.0
29    2018-02-02 0:29    8.2
...                ...    ...
3391  2018-02-04 8:31    8.0
3392  2018-02-04 8:32    8.2
3393  2018-02-04 8:33    8.0
3394  2018-02-04 8:34    8.2
3395  2018-02-04 8:35    8.2
3396  2018-02-04 8:36    8.1
3397  2018-02-04 8:37    8.0
3398  2018-02-04 8:38    8.1
3399  2018-02-04 8:39    8.2
3400  2018-02-04 8:40    8.2
3401  2018-02-04 8:41    8.1
3402  2018-02-04 8:42    8.1
3403  2018-02-04 8:43    8.0
3404  2018-02-04 8:44    8.2
3405  2018-02-04 8:45    8.1
3406  2018-02-04 8:46    8.2
3407  2018-02-04 8:47    8.0
3408  2018-02-04 8:48    8.2
3409  2018-02-04 8:49    8.1
3410  2018-02-04 8:50    8.0
3411  2018-02-04 8:51    8.0
3412  2018-02-04 8:52    8.2
3413  2018-02-04 8:53    8.0
3414  2018-02-04 8:54    8.2
3415  2018-02-04 8:55    8.2
3416  2018-02-04 8:56    8.1
3417  2018-02-04 8:57    8.0
3418  2018-02-04 8:58    8.1
3419  2018-02-04 8:59    8.2
3420  2018-02-04 9:00    8.2

[3421 rows x 2 columns]
```

Pre-process data: parse “Timestamp” column as datetime and set it as index.

```
In [4]: processed_data = process_data(test_data)
```

Plot the test data.

```
In [5]: f1 = plt.figure(figsize=(20,10))
        test_data['Value'].plot()
        plt.xlabel("Timestamp")
        plt.ylabel("Value")
        plt.title("Test Data")
        plt.show()
```



It shows clearly there are two anomalies in the data set.

Read data from csv file.

```
In [6]: data = pd.read_csv('TestData-ESD.csv', index_col='Timestamp',
                                parse_dates=True, squeeze=True,
        date_parser=pd_parser4)
```

Detect anomalies in data set.

```
In [7]: results = detts.anomaly_detect_ts(data, max_anoms=0.01,
                                                direction='both', plot=False)
```

Display detection result.

```
In [8]: results['anoms']
```

```
Out[8]: 2018-02-02 02:02:00    7.0
        2018-02-02 20:02:00    8.9
        dtype: float64
```

S-H-ESD successfully detects all the anomalies in the data set.

2.4.3 Exponential Moving Average

In statistics, a moving average (rolling average, moving mean or running average) is a calculation that creates series of averages of different subsets of the full data set. It is generally used to analyze temporal data such as to gauge the direction of the current stock trend.

Given a sequence of N values ($\{p\}_{i=1}^{N}$), and a window size $n > 0$; an n moving average of the given sequence is a new sequence ($\{S\}_{i=1}^{N-n+1}$) that is defined from p_i by taking the arithmetic mean of subsequences of n terms,

$$S_i = \frac{1}{n} \sum_{j=i}^{i+n-1} p_j \tag{2.10}$$

Thus the n moving averages are defined as:

$$S_2 = \frac{1}{2}(p_1 + p_2, p_2 + p_3, \cdots, p_{n-1} + p_n) \tag{2.11}$$

$$S_3 = \frac{1}{3}(p_1 + p_2 + p_3, p_2 + p_3 + p_4, \cdots, p_{n-2} + p_{n-1} + p_n) \tag{2.12}$$

and so on.

Exponential moving average (EMA), also known as an exponentially weighted moving average (EWMA), places a greater weight and significance on the most recent data points rather than the old ones.

The calculation of the EMA of current point is defined as:

$$EMA_p = \alpha \times \Big(p + (1-\alpha)p_1 + (1-\alpha)^2 p_2 + (1-\alpha)^3 p_3 + \cdots\Big) \tag{2.13}$$

where:

- The coefficient α is a constant smoothing factor between 0 and 1.
- p is current point
- p_1 the previous point
- and so on so forth

Exponentially weighted moving averages react more significantly to recent value changes. The weighting for each previous data exponentially decreases but never reaches zero. Luminol uses EMA to compute exponential moving averages of derivatives.

LinkedIn Luminol Anomaly Detection and Correlation Library

LinkedIn Luminol is a Python library for time series data analysis (LinkedIn 2015). It supports anomaly detection as well as correlation and can be used to investigate possible root causes for anomalies as well. With collected time series data, Luminol is able to:

- Detect anomalies in temporal data and return a time range to indicate when the anomalies occurred and a timestamp when the anomaly exactly happened (reached the large severity), and a score to show how severe the anomaly was in regards to other data.
- Determine correlation coefficient in two time series data as the correlation mechanism allows for a shift, users are able to correlate two peaks that are apart in time.

Luminol is configurable so that users can choose which algorithm they want to use for anomaly detection or correlation. In addition, the library does not rely on any predefined threshold on the values of a time series. Instead, it assigns each data point an anomaly score and identifies anomalies using the scores (LinkedIn 2015).

By using the Luminol, users would be able to establish a logic flow for the root cause analysis of anomalies. For instance, suppose there is a spike in network latency:

- Luminol discovers the spike in time series data of network latency
- gets the anomaly period of that spike, and then correlates that with other system metrics such as IO, CPU within the same time range
- gets a ranked list of correlated metrics, and the ones with high rank on the list could be the root cause candidates

2.4.4 Example for EMA

The following Jupyter Notebook shows how to use EMA for anomaly detection. This example uses test data which are listed in Table 2.2. In the data set, the value "1" at "2018/1/1 0:15" is the anomaly. The following example shows that EMA successfully finds this anomaly.

Run the reusable Python code for library importing and auxiliary functions. Details of this file is in Section 3.2.

```
In [1]: %run anomaly_detection.py
```

Initialize the detector instance using test data and choose Exponential Moving Average Algorithm.

```
In [2]: detector = AnomalyDetector('TestData-LinkedIn.csv',
                                   algorithm_name='exp_avg_detector')
```

Get the anomalies of test data set.

```
In [3]: result = detector.get_anomalies()
```

Display the anomalies. Timestamp in the result is epoch, so we need to convert it to UTC data time string.

```
In [4]: anomalies = format_luminol_result(result, False)

In [5]: anomalies

Out[5]:    Number             Start On               End On    Exactly Happen On
        0       1  2018-01-01 00:15:00  2018-01-01 00:15:00  2018-01-01 00:15:00
```

The result of Luminol actually gives the time when the anomalies start, the time when the anomalies end, and the exact time when the anomaly happens.

Chapter 3

Experiments

3.1 Anomaly Detection Techniques

In this chapter, the techniques discussed in the previous chapter have been evaluated thoroughly using Jupyter Notebooks:

- Machine Learning
  - Local Outlier Factor
- Statistic Techniques
  - Seasonal Hybrid Extreme Studentized Deviate
  - Exponential Moving Average

Two separated experiments have been performed for the evaluation.

- Experiment I (Section 3.3) evaluates these three techniques on two data sets of water temperature and ammonia data. Each data set has over 70,000 records (data details are in Section 3.3.1).
- Experiment II (Section 3.4) evaluates these three techniques on four data sets of water temperature, ammonia, chloride and potassium data. Each data set has nearly 300,000 records (data details are in Section 3.4.1).

Experiment I is the primary evaluation and Experiment II was used for verification.

3.2 Generic Python Code for Experiment

All auxiliary functions that are called in Jupyter Notebook are predefined in a separate Python file which was run at the beginning. These functions include reading data from a csv file, data pre-process and result plotting. The following is the reusable Python code of this file.

```
1 import pandas as pd
2 import numpy as np
3 import math
4 import time
5 import matplotlib.pyplot as plt
6 from sklearn.base import TransformerMixin
```

```
from sklearn.pipeline import Pipeline
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import MinMaxScaler
from sklearn.neighbors import LocalOutlierFactor
import matplotlib.pylab as pylab

#Impor LinkedIn luminol
import luminol
from luminol import utils, anomaly_detector
from luminol.anomaly_detector import AnomalyDetector

#Twitter AnomalyDetect
import anomaly_detect_ts as detts

#Date Time Parser
def pd_parser(date):
    return pd.datetime.strptime(date, '%Y-%m-%d %H:%M:%S')

#Date Time Parser
def pd_parser2(date):
    return pd.datetime.strptime(date, '%Y/%m/%d %H:%M')

#Date Time Parser
def pd_parser3(date):
    return pd.datetime.strptime(date, '%m/%d/%Y %H:%M')

#Now we create two methods: set_col_as_index is a method to set column as index
    and sort_time_series is a method to sort data based on the date time series
    data. They will be used later on to process and transform data to the index by
     time data column format.
#set index column
class set_col_as_index(TransformerMixin):
    def __init__(self, col):
        self.col = col

    def transform(self, X, **transform_params):
        X.index = X.loc[:, self.col].apply(lambda x: pd.to_datetime(x))
        return X

    def fit(self, X, y=None, **fit_params):
        return self

#sort data based on the index
class sort_time_series(TransformerMixin):
    def transform(self, X, **transform_params):
        X = X.sort_index()
        return X

    def fit(self, X, y=None, **fit_params):
        return self

#A common method which is used to create index and transform data the 'TimeStamp'
     collomn
def process_data(x):
    process_pipeline = make_pipeline(set_col_as_index('Timestamp'),
                                     sort_time_series())
```

```
60      x = process_pipeline.fit_transform(x)
61      del x['Timestamp']
62      return x
63
64  #At the beginning, we create methods to get data from csv files, process and then
        display them.
65  #Belowing method get_temp_data is to read water temperature test data from csv
        file.
66  def get_temp_data():
67      return pd.read_csv("Temp.csv")
68
69  #Method get_ammonia_data is to read water ammonia test data from csv file.
70  def get_ammonia_data():
71      return pd.read_csv("Ammonia.csv")
72
73  #display the general information of dataset
74  def display_info(df):
75      print ("Data Range")
76      print ("Start Date %s"%(df.head(1)['Timestamp']))
77      print ("End Date %s"%(df.tail(1)['Timestamp']))
78      print ("num_values: %s"%(df.shape[0]))
79
80  #Plot the orginal data set
81  def plot_original_data(df,title):
82      f1 = plt.figure(figsize=(20,10))
83      df['Value'].plot()
84      plt.xlabel("Timestamp")
85      plt.ylabel("Value")
86      plt.title(title)
87      plt.show()
88
89  #prepare traning dataset
90  def prepare_traning_dataset(df):
91      #get value of dataset
92      X = df['Value'].values
93      #Then change the precision of value to 4.
94      #X= np.round(X,4)
95      #Reshape data to only one column
96      X = X.reshape(-1, 1)
97      return X
98
99  #prepare traning dataset
100 def reshape_traning_dataset(df):
101     #get value of dataset
102     X = df['Value'].values
103     #Then change the precision of value to 2.
104     X= np.round(X,2)
105     #Reshape data to only one column
106     X = X.reshape(-1, 1)
107     return X
108
109 def lof_prediction(clf,training_data_set, data_set):
110     #fit clf with training dataset
111     y_pred = clf.fit_predict(training_data_set)
112     #Get the prediction score
113     y_pred = clf._decision_function(training_data_set)
```

```
114     #Combine test data and prediction result as the prediction only contains
        value and score.
115     data_set['isinlier'] = y_pred
116     #Sort combined data by result score, lower score indicates the higher
        potential to be an outlier.
117     #We consider those data score less than -10 are outliers (threshold)
118     #Get the data whose score are less than -10 from the prediction result.
119     anomalies = data_set.loc[data_set['isinlier'] < -10]
120     data_set
121     return anomalies
122 
123 #Method to plot the result of LOF anomaly detection
124 def plot_lof_result(dataset,anomalies,title):
125     if 'isinlier' in dataset.columns:
126         del dataset['isinlier']
127     if 'isinlier' in anomalies.columns:
128         del anomalies['isinlier']
129     f2 = plt.figure(figsize=(20,10))
130     plt.plot(dataset, color='green')
131     plt.plot(anomalies,"r*", markersize=10)
132     plt.xlabel("Timestamp")
133     plt.ylabel("Value")
134     plt.title(title)
135     plt.show()
136 
137 #Method 'get_anormalies(y, clf)' is to score each data using Local Outlier Factor
        inside subset and return those whose score is less than -10 (threshold which
        we consider is anomaly).
138 #Parameters:
139 #y: Subset of test data set
140 #clf: Instance of Class LocalOutlierFactor of Scikit-learn library
141 #return: Anomalies
142 def get_anormalies(y, clf):
143     y.is_copy = False
144     y_pred = clf._decision_function(y)
145     y['isinlier'] = y_pred
146     return y.loc[y['isinlier'] <-10 ]
147 
148 #Method 'calculate_outlier(x, window_size, clf)' is used to get the outlier of
        the data set by dividing it into subsets. It calls function 'get_anormalies'
        to make decision on each subset and return the accumulated anomalies.
149 #Parameters:
150 #x: the data set
151 #window_size: The size of each subset
152 #clf: Instance of Class LocalOutlierFactor of Scikit-learn library
153 def calculate_outlier(x, window_size, clf):
154     length = x.shape[0]
155     result =  pd.DataFrame(columns = x.columns)
156     i = 0
157     while  i< length:
158         begin_index = i
159         i += window_size
160         end_index = i
161         if( i > length):
162             end_index = length
163         y = x.iloc[begin_index:end_index]
164         y.is_copy = False
```

```
165         yield get_anormalies(y, clf)
166
167 #Method to display result for luminol
168 #return list of anomalies
169 def format_luminol_result(anomalies, display):
170     rows_list=[]
171     i=1
172     for timestamp in anomalies:
173         start = time.localtime(timestamp.start_timestamp/1000)
174         start_str = time.strftime('%Y-%m-%d %H:%M:%S',start)
175         end =time.localtime(timestamp.end_timestamp/1000)
176         end_str = time.strftime('%Y-%m-%d %H:%M:%S', end )
177         exact = time.localtime(timestamp.exact_timestamp/1000)
178         ex_str = time.strftime('%Y-%m-%d %H:%M:%S', exact)
179         rows_list.append([ i,start_str, end_str,ex_str])
180         if display:
181             print("Anomaly %s:"%(i))
182             print ("Start: %s"%(start_str))
183             print ("End: %s"%(end_str))
184             print ("Exact: %s"%(ex_str))
185         i= i+1
186     return pd.DataFrame(rows_list,
187                         columns=['Number', 'Start On',
188                                  'End On', 'Exactly Happen On'])
```

3.3 Evaluation using Jupyter Notebook I

3.3.1 Test Data Sets

To evaluate the performance of each algorithm, two data sets from the RSM30 water monitoring system were used in this experiment. These data sets were provided by Primodal System and retrieved from the Dundas wastewater treatment plant.

Table 3.1 summarizes the properties of the water temperature data set.

File Name	temp.csv
Location	Dundas Treatment Facility
Sensor model	IQ Sensor Net Varion@ Plus 700 IQ
Interval	Every minute
Number of Records	78338
Columns	Timestamp and Value
Date Range	2017/11/1 14:20:00 to 2018/1/9 19:55:00

TABLE 3.1: Water Temperature Data

Figure 3.1 illustrates data for water temperature.

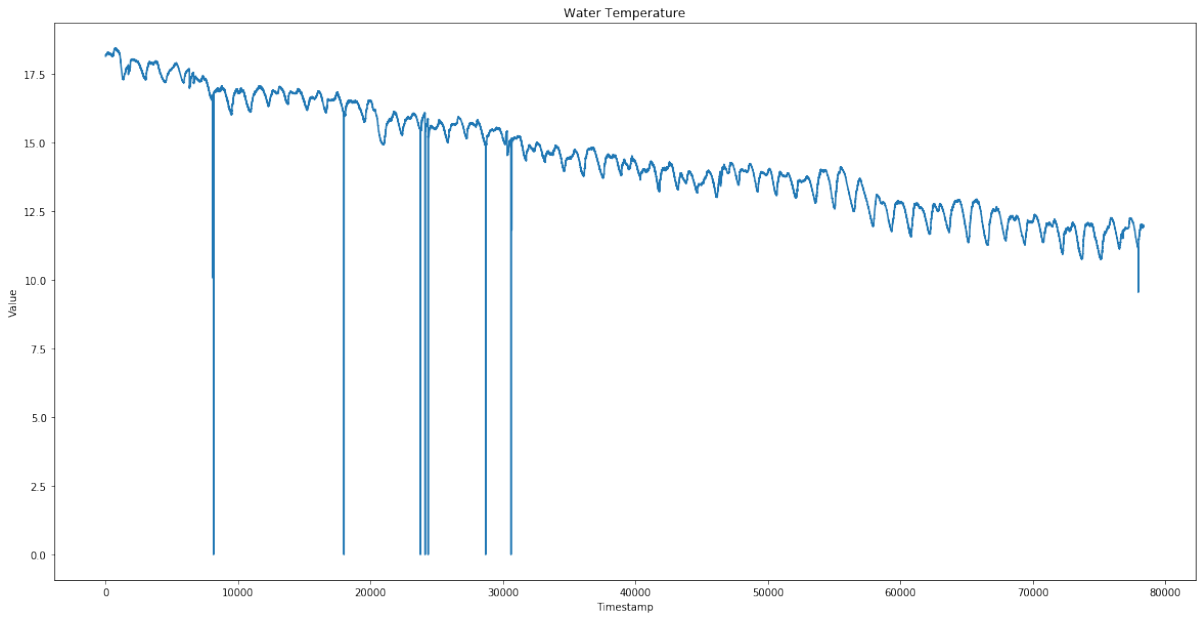


FIGURE 3.1: Temperature Data Set

Table 3.2 summarizes the properties of the ammonia data set.

File Name	ammonia.csv
Location	Dundas Treatment Facility
Sensor model	IQ Sensor Net Varion@ Plus 700 IQ
Interval	Every minute
Number of Instances	71926
Columns	Timestamp and Value
Date range	2017/12/05 19:00:00 to 2018/1/26 18:07:00

TABLE 3.2: Ammonia Data

Figure 3.2 illustrates data for ammonia.

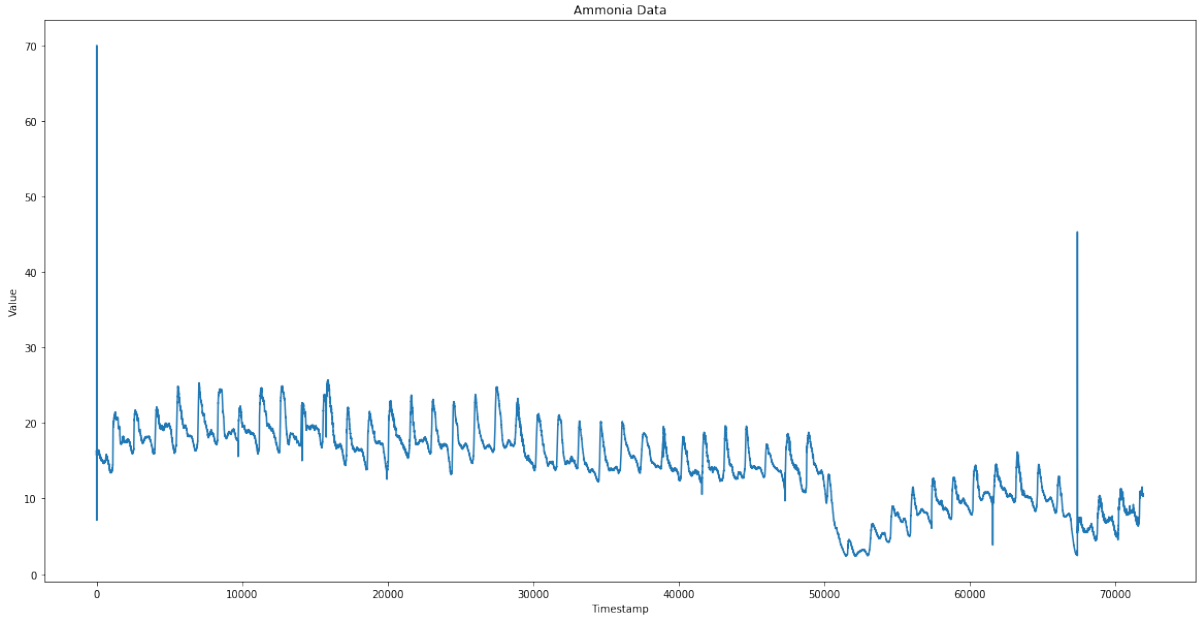


FIGURE 3.2: Ammonia Data Set

3.3.2 Notebook

Machine Learning

First of all, the unsupervised outlier detection class 'LocalOutlierFactor' in scikit-learn was used to detect outliers in water temperature data and ammonia data.

Run Python file for the auxiliary functions.

```
In [1]: %run anomaly_detection.py
```

Read temperature data from "Temp.csv" file and save to df_temp. "Temp.csv" contains two columns: "Timestamp" and "Value".

```
In [2]: df_temp = get_temp_data()
```

Then display the general information of the test temperature data.

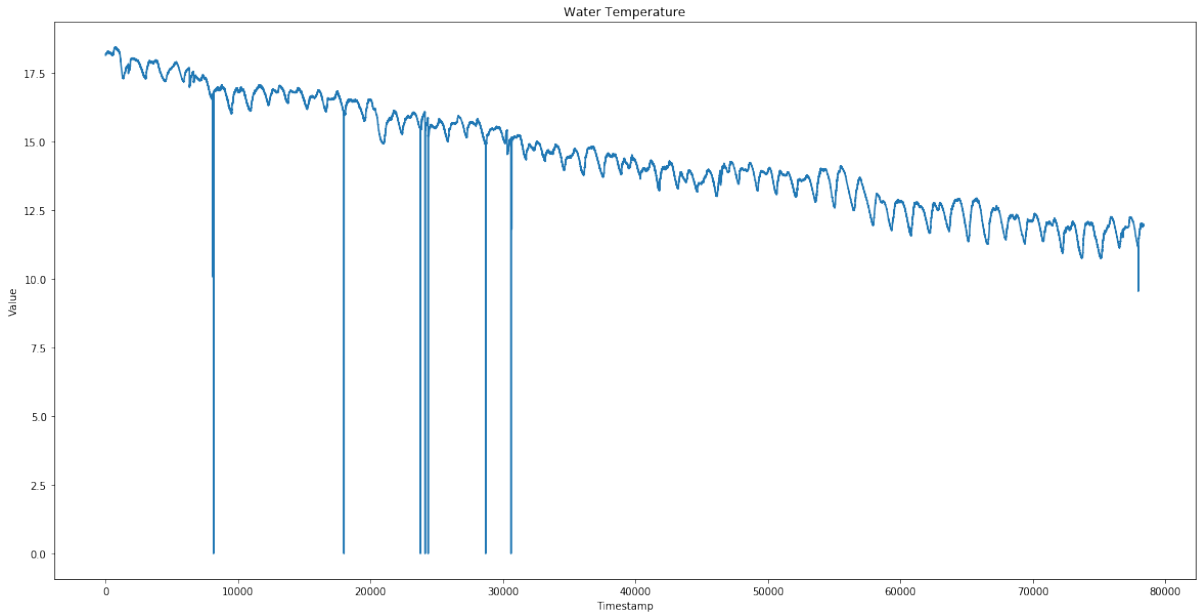
```
In [3]: display_info(df_temp)
```

```
Data Range
Start Date 0    2017/11/1 14:20
Name: Timestamp, dtype: object
End Date 78338    2018/1/9 19:55
Name: Timestamp, dtype: object
num_values: 78339
```

The data set has 78339 records in total.

Now plot temperature test data.

```
In [4]: plot_original_data(df_temp, "Water Temperature")
```



The above output diagram clearly shows that there are several instances which are far from the others which probably indicate point anomalies.

The following steps get ammonia test data from csv file, process and then display it.

Call function "get_ammonia_data" to retrieve ammonia test data from "Ammonia.csv" file. "Ammonia.csv" contains two columns: "Timestamp" and "Value".

```
In [5]: df_ammonia = get_ammonia_data()
```

Display general information of anomaly test data.

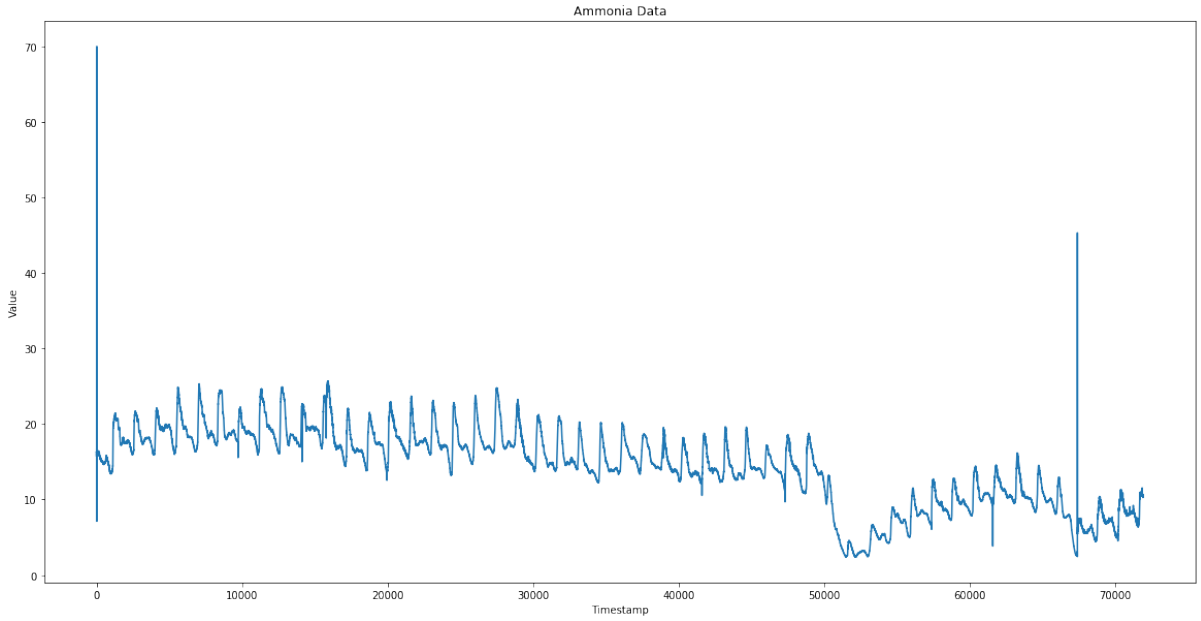
```
In [6]: display_info(df_ammonia)
```

```
Data Range
Start Date 0    2017-12-05 19:00:00
Name: Timestamp, dtype: object
End Date 71925    2018-01-26 18:08:00
Name: Timestamp, dtype: object
num_values: 71926
```

There are 71926 records in the ammonia data:

And then plot ammonia test data.

```
In [7]: plot_original_data(df_ammonia, "Ammonia Data")
```



Outliers have been detected using Local Outlier Factor in the scikit-learn library.

Prepare training data set of temperature values.

Function "process_data" creates index on "Timestamp" column for temperature data.

```
In [8]: df_temp = process_data(df_temp)
```

Function "prepare_training_dataset" gets data of "Value" column which is used to fit LOF class later.

```
In [9]: temp_data = prepare_training_dataset(df_temp)
```

Prediction of test data is done by an instance of LocalOutlierFactor class. The constructor of LocalOutlierFactor has several parameters:

- n_neighbors : The value for k. The default value is 20.
- metric : The metric used for the distance computation.

As discussed in Section 2.3.5, k in range 10 to 50 generally works well. n_neighbors = 50 is used for k, and euclidean distance is used to measure the distance. Other distance metrics have been discussed in Section 2.3.5.

```
In [10]: clf = LocalOutlierFactor(n_neighbors=50, metric='euclidean')
```

Get the prediction result of temperature test data.

```
In [11]: %%time
         temp_anomalies = lof_prediction(clf,temp_data, df_temp)
```

```
CPU times: user 745 ms, sys: 122 ms, total: 867 ms
Wall time: 890 ms
```

```
In [12]: print ("Number of Anomalies:%s"%(temp_anomalies.shape[0]))
```

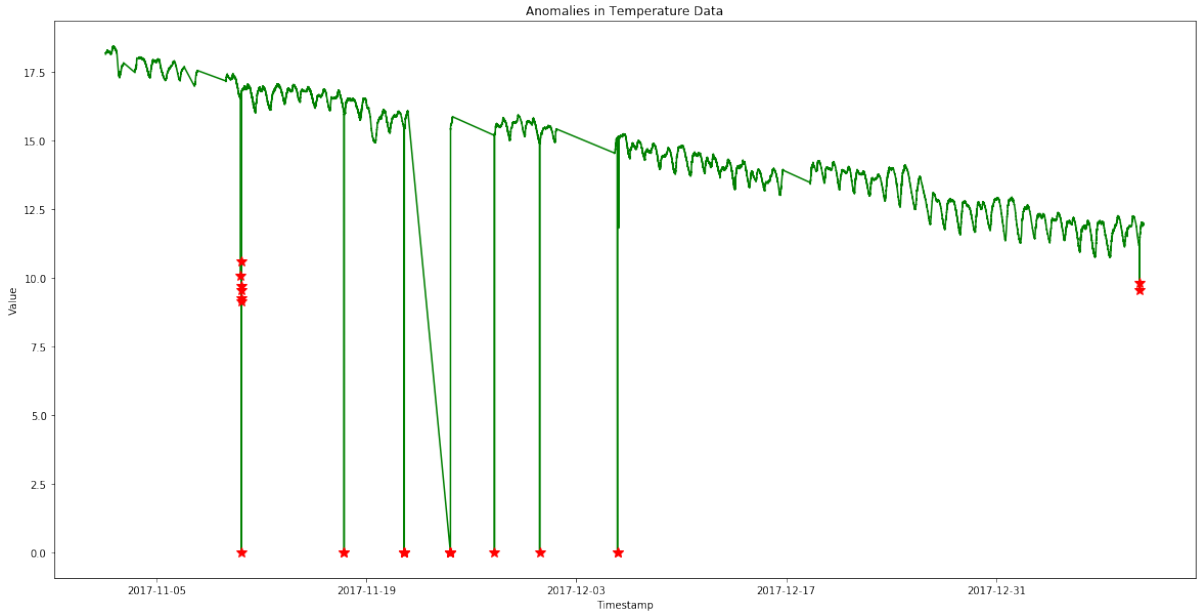
```
Number of Anomalies:26
```

Save all the abnormal instances with their score to a csv file for further analysis.

```
In [13]: temp_anomalies.to_csv('temp_anomalies.csv')
```

Plot the prediction result for temperature test data to deliver the whole picture.

```
In [14]: plot_lof_result(df_temp,temp_anomalies,'Anomalies in Temperature Data')
```



The red stars in the above output diagram are the twenty-six instances detected as anomalies. These twenty-six instances are all anomalies and are discussed in detail in Chapter 4.

Next, try to detect anomalies on the ammonia test data using the same approach.

Set index on "Timestamp" column for ammonia test data.

```
In [15]: df_ammonia  = process_data(df_ammonia )
```

Function "prepare_training_dataset" returns data of "Value" column which is used to fit LOF class later.

```
In [16]: amm_data = prepare_training_dataset(df_ammonia )
```

Initialize an instance of LocalOutlierFactor class.

```
In [17]: clf = LocalOutlierFactor(n_neighbors=50, metric='euclidean')
```

Get the prediction result.

```
In [18]: %%time
         amm_anomalies = lof_prediction(clf,amm_data, df_ammonia)

CPU times: user 727 ms, sys: 11.5 ms, total: 738 ms
Wall time: 785 ms
```

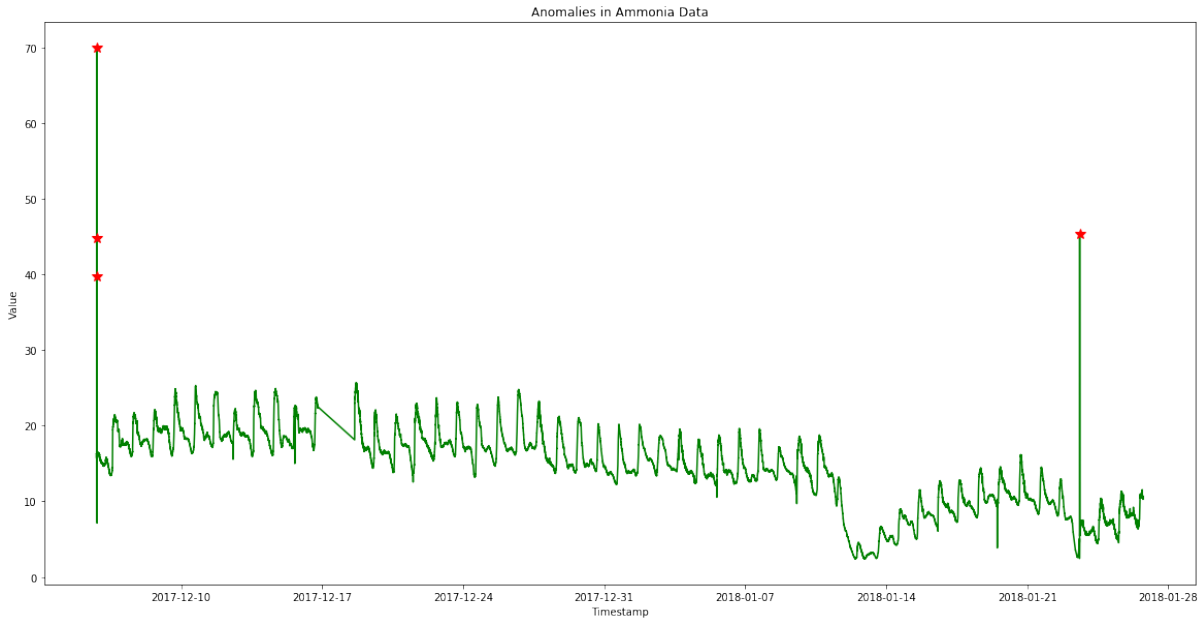
```
In [19]: print ("Number of anomalies:%s"%(amm_anomalies.shape[0]))

Number of anomalies:4
```

Save all the anomal instances with their score to a csv file for further analysis.

```
In [20]: df_ammonia.to_csv('ammonia_anomalies.csv')
```

Plot the prediction result for ammonia test data.

```
In [21]: plot_lof_result(df_ammonia,amm_anomalies,'Anomalies in Ammonia Data')
```



The red stars in the above output diagram are the four instances detected as anomalies. These four instances are all anomalies and are discussed in detail in Chapter 4.

Statistic Approach

A Python library which rewrites Twitter's AnomalyDetection algorithms is used here for testing purposes.

Read data from Temp.csv file, parse "Timestamp" column using date time parser and set it as index column.

```
In [22]: data = pd.read_csv('Temp.csv', index_col='Timestamp',
                                parse_dates=True, squeeze=True,
         date_parser=pd_parser2)
```

Function anomaly_detect_ts is used to detect anomalies for seasonal univariate time series data. The following are its parameters:

- x: Time series data which has two columns. The first column is timestamp data and the second column contains values.
- max_anoms: Maximum percentage of the data that S-H-ESD will detect as anomalies.
- max_anoms: Maximum percentage of the data that S-H-ESD will detect as anomalies.
- direction: Anomalies direction. There are three options: 'pos' (positive), 'neg' (negative) and 'both'.
- plot: A Boolean value to indicate if a plot needs to return for test data and anomalies. It is not implemented yet.

```
In [23]: %%time
         results = detts.anomaly_detect_ts(data, max_anoms=0.005,
                                                 direction='both', plot=False)

CPU times: user 3.65 s, sys: 51.4 ms, total: 3.7 s
Wall time: 3.79 s
```

Display the result.

```
In [24]: results['anoms']

Out[24]: 2017-11-24 14:20:00    0.0
         dtype: float64
```

Only one anomaly has been found.

Detect on ammonia data.

```
In [25]: data = pd.read_csv('Ammonia.csv', index_col='Timestamp',
                                parse_dates=True, squeeze=True,
         date_parser=pd_parser)

In [26]: %%time
         results = detts.anomaly_detect_ts(data, max_anoms=0.005,
                                                 direction='both', plot=False)
```

```
CPU times: user 3.15 s, sys: 0 ns, total: 3.15 s
Wall time: 3.25 s
```

Display the test result.

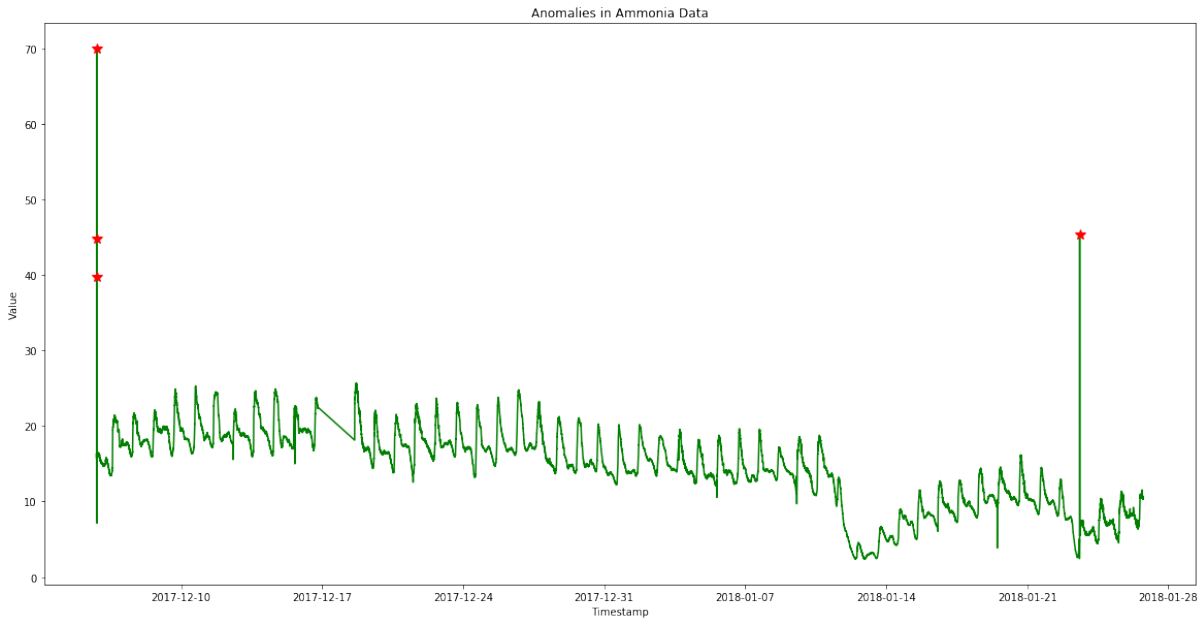
```
In [27]: results['anoms']
```

```
Out[27]: 2017-12-05 19:44:00    69.996399
         2018-01-23 14:30:00    45.311798
         2017-12-05 19:43:00    44.756802
         2017-12-05 19:42:00    39.700600
         dtype: float64
```

A total of four outliers were found for ammonia which is the same as what was found using LOF.

Plot the result.

```
In [28]: amm_plot = plt.figure(figsize=(20,10))
         plt.plot(data, color='green')
         plt.plot(results['anoms'],"r*", markersize=10)
         plt.xlabel("Timestamp")
         plt.ylabel("Value")
         plt.title("Anomalies in Ammonia Data")
         plt.show()
```



Next, try LinkedIn Python library for time series data analysis.

Initialize the detector instance using test data and choose the Exponential Moving Average algorithm which is discussed in Section 2.4.3.

```
In [29]: %%time
         detector = AnomalyDetector('TempLinkedIn.csv', algorithm_name='exp_avg_detector')
```

```
CPU times: user 18 s, sys: 300 ms, total: 18.3 s
Wall time: 20 s
```

Get the anomalies of test data set.

```
In [30]: anomalies = detector.get_anomalies()
```

Display the anomalies. Timestamp in the result is epoch, convert it to UTC date time string.

```
In [31]: temp_anomalies = format_luminol_result(anomalies, False)
```

```
In [32]: temp_anomalies
```

Out[32]:

	Number	Start On	End On	Exactly Happen On
0	1	2017-11-10 15:45:00	2017-11-10 15:45:00	2017-11-10 15:45:00
1	2	2017-11-17 12:08:00	2017-11-17 12:09:00	2017-11-17 12:09:00
2	3	2017-11-21 12:26:00	2017-11-21 12:29:00	2017-11-21 12:29:00
3	4	2017-11-21 12:31:00	2017-11-21 12:32:00	2017-11-21 12:32:00
4	5	2017-11-24 14:19:00	2017-11-24 14:22:00	2017-11-24 14:22:00
5	6	2017-11-24 14:25:00	2017-11-24 14:26:00	2017-11-24 14:26:00
6	7	2017-11-27 12:52:00	2017-11-27 12:52:00	2017-11-27 12:52:00
7	8	2017-11-30 13:13:00	2017-11-30 13:13:00	2017-11-30 13:13:00
8	9	2017-12-05 17:39:00	2017-12-05 17:39:00	2017-12-05 17:39:00
9	10	2017-12-05 17:47:00	2017-12-05 17:47:00	2017-12-05 17:47:00

The result of Luminol actually gives the start time of the anomaly and the end time of the anomaly as well.

Test Luminol on ammonia data following the above steps.

```
In [33]: %%time
         detector = AnomalyDetector('AmmoniaLinkedIn.csv',
                                    algorithm_name='exp_avg_detector')
```

```
CPU times: user 16.3 s, sys: 239 ms, total: 16.5 s
Wall time: 17.4 s
```

```
In [34]: anomalies = detector.get_anomalies()
```

Display the anomalies. Timestamp in the result is epoch, now convert it to UTC date time string.

```
In [35]: ammonia_anomalies = format_luminol_result(anomalies, False)
```

Display the results.

```
In [36]: ammonia_anomalies
```

Out[36]:

	Number	Start On	End On	Exactly Happen On
0	1	2017-12-05 19:42:00	2017-12-05 19:45:00	2017-12-05 19:44:00
1	2	2018-01-23 14:30:00	2018-01-23 14:30:00	2018-01-23 14:30:00

Two anomaly periods were detected. They cover all 4 anomalies found by other algorithms.

Isolation Forest

In this section, the standard isolation forest is compared with the extended isolation forest using the EIF package. Then, detect anomalies in the temperature data and ammonia data using the scikit-learn implementation of the isolation forest algorithm.

```
In [1]: %matplotlib inline
        import matplotlib.pyplot as plt
        import numpy as np
        from scipy.stats import multivariate_normal
        import random as rn
        import eif as iso
        import seaborn as sb
        sb.set_style(style="whitegrid")
        sb.set_color_codes()
        import scipy.ndimage
        from scipy.interpolate import griddata
        import numpy.ma as ma
        from numpy.random import uniform, seed
        import pandas as pd
```

These two functions are used to find the depth a given data point reaches in an IF tree.

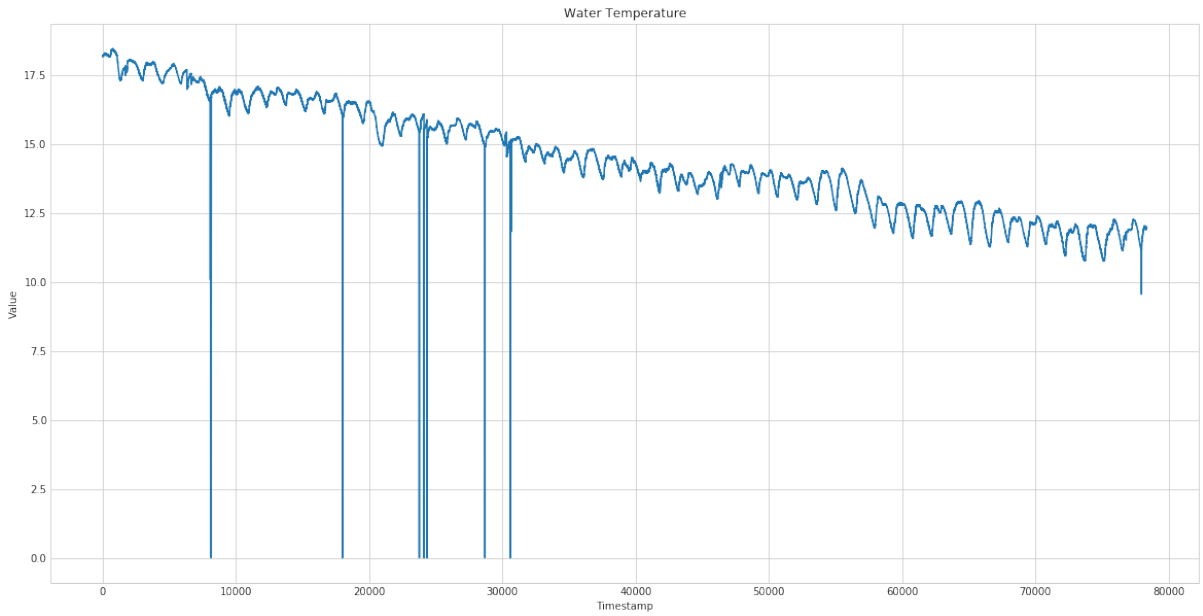
```
In [2]: def getDepth(x, root, d):
            n = root.n
            p = root.p
            if root.ntype == 'exNode':
                return d
            else:
                if (x-p).dot(n) < 0:
                    return getDepth(x,root.left,d+1)
                else:
                    return getDepth(x,root.right,d+1)

        def getVals(forest,x, sorted=True):
            theta = np.linspace(0,2*np.pi, forest.ntrees)
            r = []
            for t in forest.Trees:
                r.append(getDepth(x,t.root,1))
            if sorted:
```

```
            r = np.sort(np.array(r))
        return r, theta
```

Load water temperature data from csv file.

```
In [3]: df_temp =  pd.read_csv("Temp.csv")
```

```
In [4]: f1 = plt.figure(figsize=(20,10))
        df_temp['Value'].plot()
        plt.xlabel("Timestamp")
        plt.ylabel("Value")
        plt.title('Water Temperature')
        plt.show()
```



Change timestamp to epoch.

```
In [5]: epoch = pd.to_datetime(df_temp['Timestamp']).values.astype(np.int64)
```

```
In [6]: df_temp['Timestamp'] = epoch
```

Convert dataframe to array.

```
In [7]: X = np.array(df_temp)
```

Set values of axis x for plotting.

```
In [8]: x = np.array(df_temp['Timestamp'])
```

Set values of axis y for plotting.

```
In [9]: y = np.array(df_temp['Value'])
```

Two sets of forests, F0 and F1 are trained for comparison.

F0 is the standard Isolation Forest, which corresponds to extension level 0 in the context of EIF.

```
In [10]: %%time
         #ExtensionLevel=0 is the same as regular Isolation Forest
         F0  = iso.iForest(X , ntrees=100, sample_size=256, ExtensionLevel=0)
```

```
CPU times: user 770 ms, sys: 8.97 ms, total: 779 ms
Wall time: 921 ms
```

F1 is the Extended Isolation Forest with extension 1, which is the fully extended case.

```
In [11]: %%time
         F1  = iso.iForest(X , ntrees=100, sample_size=256, ExtensionLevel=1)
```

```
CPU times: user 920 ms, sys: 16.7 ms, total: 936 ms
Wall time: 1.03 s
```

Function compute_paths returns that anomaly score. It computes the depth each points reaches in each trained tree, and converts the ensemble aggregate to an anomaly score.

```
In [12]: %%time
         # Score the training data itself to see the distribution of
         #the anomaly scores each point receives.
         S0 = F0.compute_paths(X_in=X)
```

```
CPU times: user 2min 54s, sys: 14.7 ms, total: 2min 54s
Wall time: 2min 56s
```

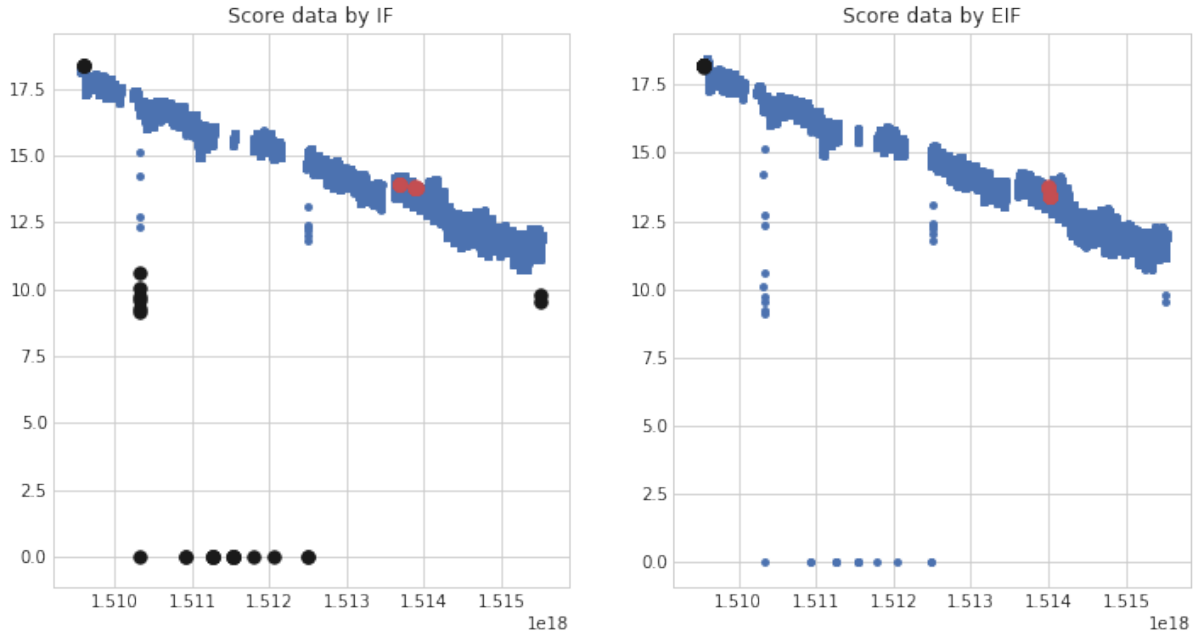
```
In [13]: %%time
         S1 = F1.compute_paths(X_in=X)
```

```
CPU times: user 2min 49s, sys: 47.1 ms, total: 2min 49s
Wall time: 3min 3s
```

Plot the points and highlight 30 points with highest and 30 points with lowest anomaly scores. The two plots provide a comparison between the two algorithms.

```
In [14]: ss0=np.argsort(S0)
         ss1=np.argsort(S1)

         f = plt.figure(figsize=(12,6))
         plt.subplot(1,2,1)
         plt.scatter(x,y,s=15,c='b',edgecolor='b')
         plt.scatter(x[ss0[-30:]],y[ss0[-30:]],s=55,c='k')
         plt.scatter(x[ss0[:30]],y[ss0[:30]],s=55,c='r')
         plt.title('Score data by IF')
```

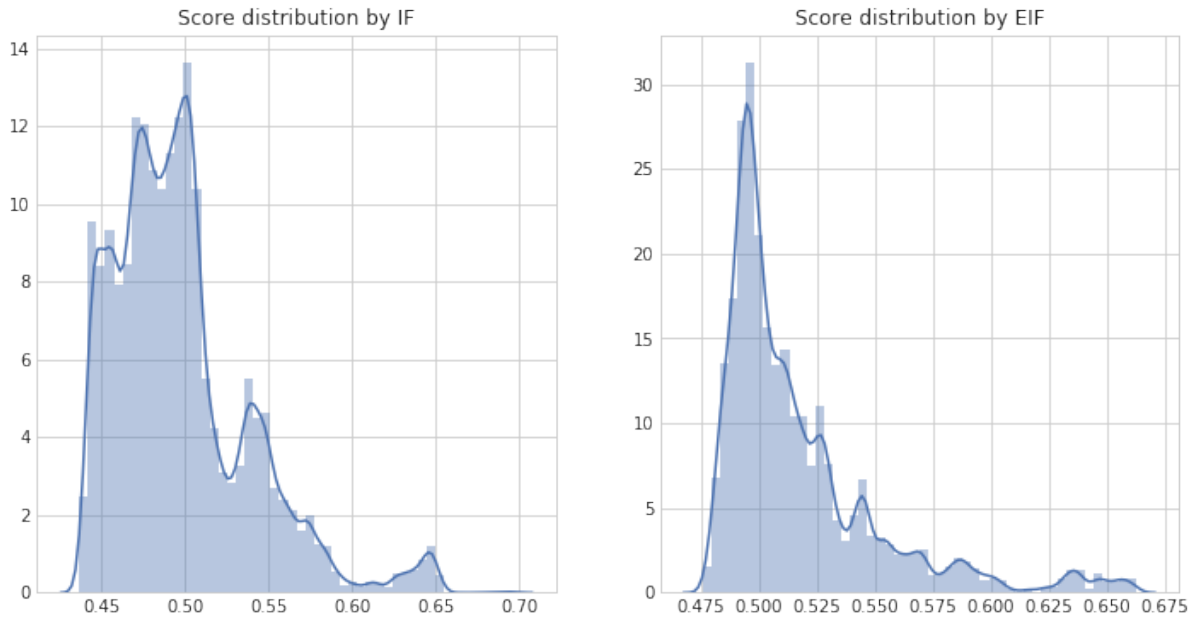
```
        plt.subplot(1,2,2)
        plt.scatter(x,y,s=15,c='b',edgecolor='b')
        plt.scatter(x[ss1[-30:]],y[ss1[-30:]],s=55,c='k')
        plt.scatter(x[ss1[:30]],y[ss1[:30]],s=55,c='r')
        plt.title('Score data by EIF')
        plt.show()
```



The distribution of anomaly scores are shown above. By definition, anomalies are those that occur less frequently. So it makes sense that the number of points with higher anomaly scores reduces as the score increases.

The plotting above shows that standard IF works better than EIF and found more anomalies. Some point anomalies were missed by EIF.

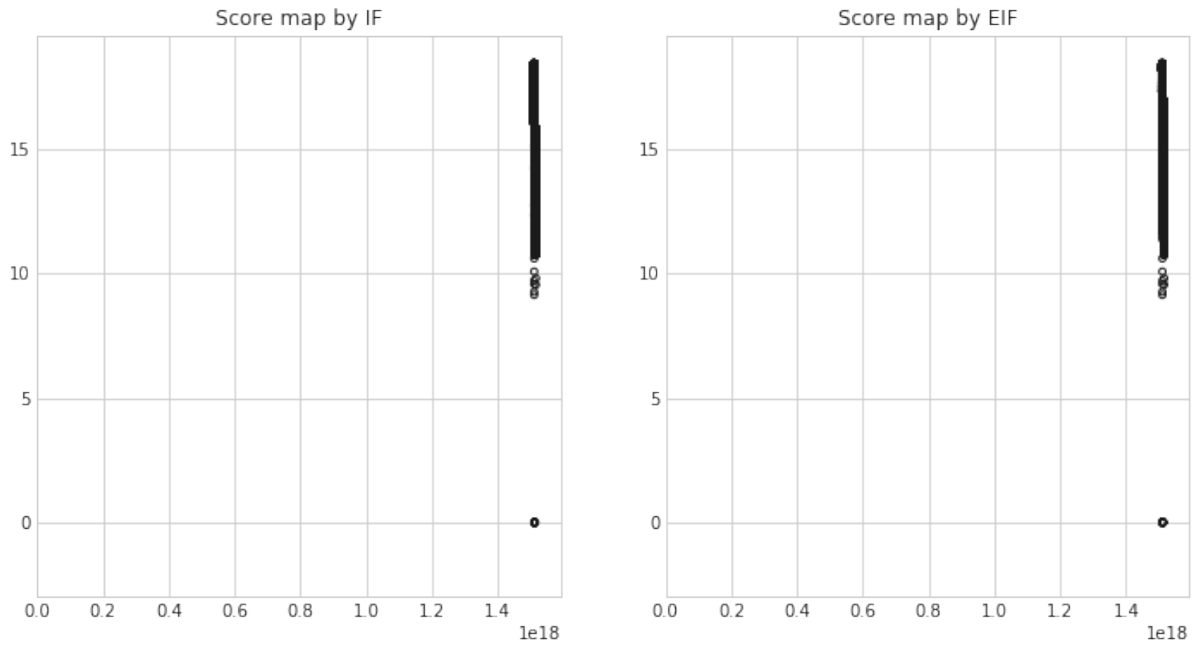
```
In [15]: f = plt.figure(figsize=(12,6))
         plt.subplot(1,2,1)
         sb.distplot(S0, kde=True, color="b")
         plt.title('Score distribution by IF')

         plt.subplot(1,2,2)
         sb.distplot(S1, kde=True, color="b")
         plt.title('Score distribution by EIF')
         plt.show()
```



A meshgrid is created on a square domain. Each point on the grid is then scored using the trained forests. The resulting score map is visualized using contour plots.

```
In [16]: xx, yy = np.meshgrid(np.linspace(-5, 30, 30), np.linspace(-3, 3, 30))

         S0 = F0.compute_paths(X_in=np.c_[xx.ravel(), yy.ravel()])
         S0 = S0.reshape(xx.shape)

         S1 = F1.compute_paths(X_in=np.c_[xx.ravel(), yy.ravel()])
         S1 = S1.reshape(xx.shape)
```

```
In [17]: f = plt.figure(figsize=(12,6))
         ax1 = f.add_subplot(121)
         levels = np.linspace(np.min(S0),np.max(S0),10)
         CS = ax1.contourf(xx, yy, S0, levels, cmap=plt.cm.YlOrRd)
         plt.scatter(x,y,s=15,c='None',edgecolor='k')
         plt.title('Score map by IF')

         ax2 = f.add_subplot(122)
         levels = np.linspace(np.min(S1),np.max(S0),10)
         CS = ax2.contourf(xx, yy, S1, levels, cmap=plt.cm.YlOrRd)
         plt.scatter(x,y,s=15,c='None',edgecolor='k')
         plt.title('Score map by EIF')
         plt.show()
```



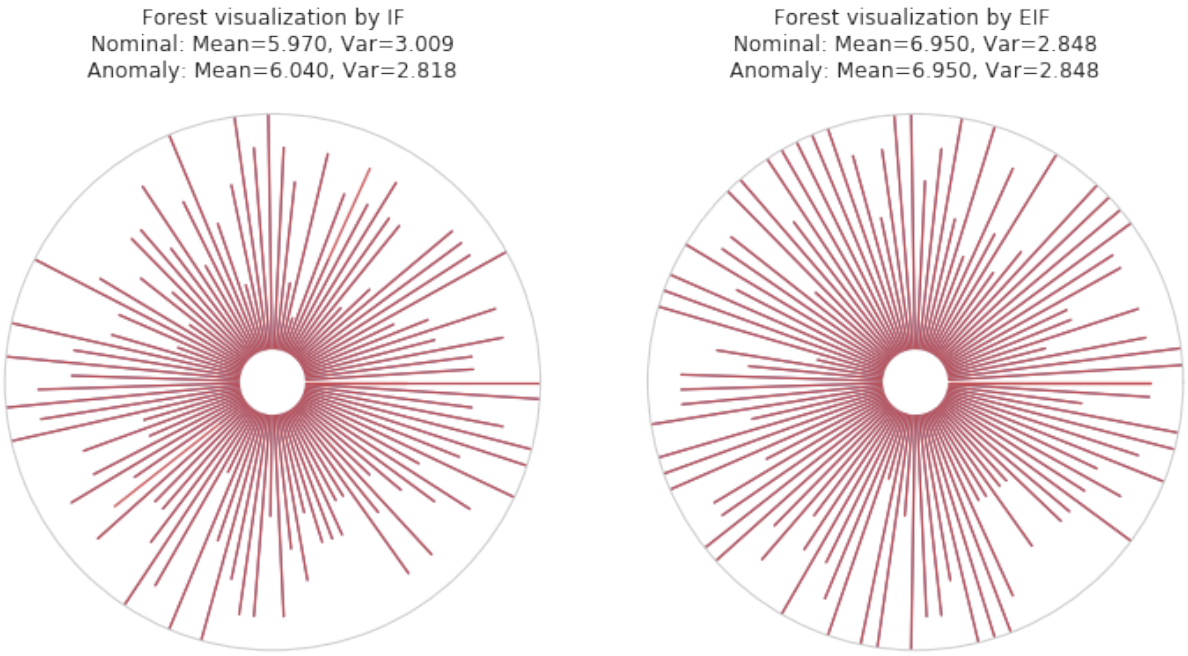
The following code is used to get a meaningful threshold to distinguish anomalies and normal data.

```
In [18]: anomalies0 = S0[S0 > 0.7]

In [19]: anomalies0.size

Out[19]: 900

In [20]: anomalies1 = S1[S1 > 0.7]

In [21]: anomalies1.size

Out[21]: 0

In [22]: anomalies0 = S0[S0 > 0.65]

In [23]: anomalies0.size

Out[23]: 900

In [24]: anomalies1 = S1[S1 > 0.65]

In [25]: anomalies1.size

Out[25]: 900
```

For each case, the forest is visualized by passing a single anomalous and a single nominal point through the forest. Each radial line in the plots below corresponds to a tree. The gray circle is the depth limit each tree can reach. Blue lines show the depth the nominal point reached on each tree, while the red lines show the depth each anomalous point reaches for each

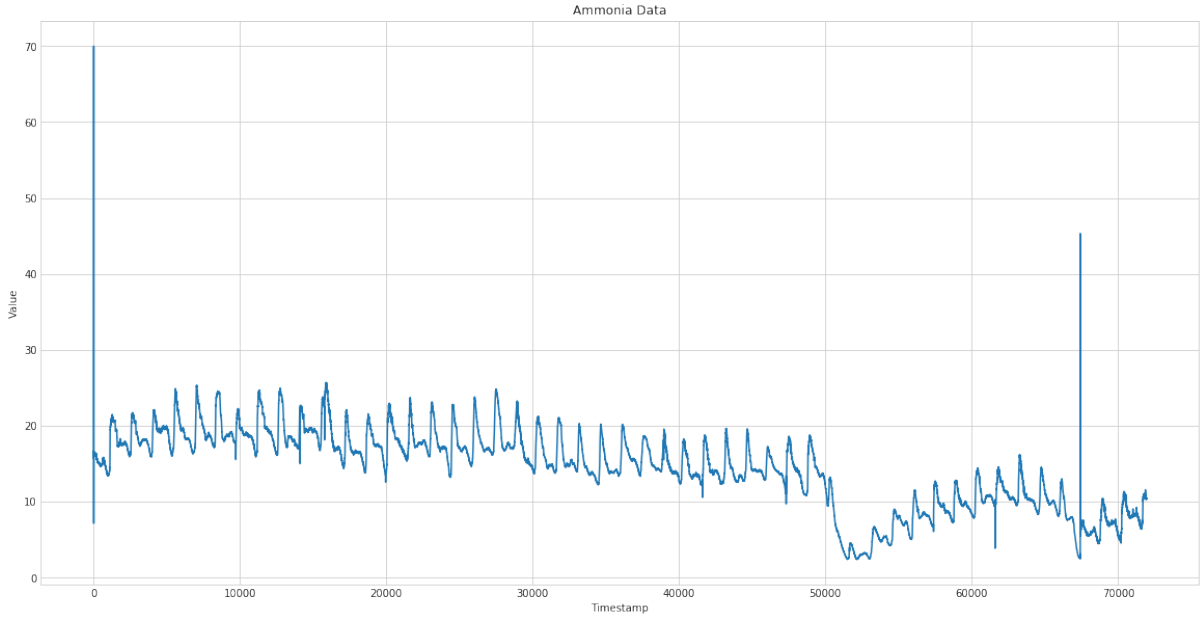
tree. This visualization provides a quick view of how an average anomalous points reaches much lower depths than the normal points.

```
In [26]: Sorted=False
         fig = plt.figure(figsize=(12,6))
         ax1 = plt.subplot(121, projection='polar')
         rn, thetan = getVals(F0,np.array([0,0]),sorted=Sorted)
         for j in range(len(rn)):
             ax1.plot([thetan[j],thetan[j]], [1,rn[j]], color='b',alpha=1,lw=1)

         ra, thetaa = getVals(F0,np.array([3.3,3.3]),sorted=Sorted)
         for j in range(len(ra)):
             ax1.plot([thetaa[j],thetaa[j]], [1,ra[j]], color='r',alpha=0.9,lw=1.3)

         title ="Forest visualization by IF\nNominal: Mean={0:.3f}, Var={1:.3f}\n" + \
         "Anomaly: Mean={2:.3f}, Var={3:.3f}"
         ax1.set_title(title.format(np.mean(rn),np.var(rn),np.mean(ra),np.var(ra)))

         ax1.set_xticklabels([])
         ax1.set_xlabel("Anomaly")
         ax1.set_ylim(0,F0.limit)

         ax1.axes.get_xaxis().set_visible(False)
         ax1.axes.get_yaxis().set_visible(False)


         ax2 = plt.subplot(122, projection='polar')
         rn, thetan = getVals(F1,np.array([0,0]),sorted=Sorted)
         for j in range(len(rn)):
             ax2.plot([thetan[j],thetan[j]], [1,rn[j]], color='b',alpha=1,lw=1)

         ra, thetaa = getVals(F1,np.array([3.3,3.3]),sorted=Sorted)
         for j in range(len(ra)):
             ax2.plot([thetaa[j],thetaa[j]], [1,ra[j]], color='r',alpha=0.9,lw=1.3)

         title ="Forest visualization by EIF \nNominal: Mean={0:.3f}, Var={1:.3f}\n" + \
         "Anomaly: Mean={2:.3f}, Var={3:.3f}"
         ax2.set_title(title.format(np.mean(rn),np.var(rn),np.mean(ra),np.var(ra)))

         ax2.set_xticklabels([])
         ax2.set_xlabel("Anomaly")
         ax2.set_ylim(0,F0.limit)

         ax2.axes.get_xaxis().set_visible(False)
         ax2.axes.get_yaxis().set_visible(False)
```



A 2-D dataset is produced with a sinusoidal shape and Gaussian noise is added on top.

```
In [27]: df_amm = pd.read_csv("Ammonia.csv")

In [28]: f1 = plt.figure(figsize=(20,10))
         df_amm['Value'].plot()
         plt.xlabel("Timestamp")
         plt.ylabel("Value")
         plt.title('Ammonia Data')
         plt.show()
```



Change timestamp to epoch.

```
In [29]: df_amm['Timestamp'] = \
         pd.to_datetime(df_amm['Timestamp']).values.astype(np.int64)
```

Set values of axis x for plotting.

```
In [30]: x = df_amm['Timestamp']
```

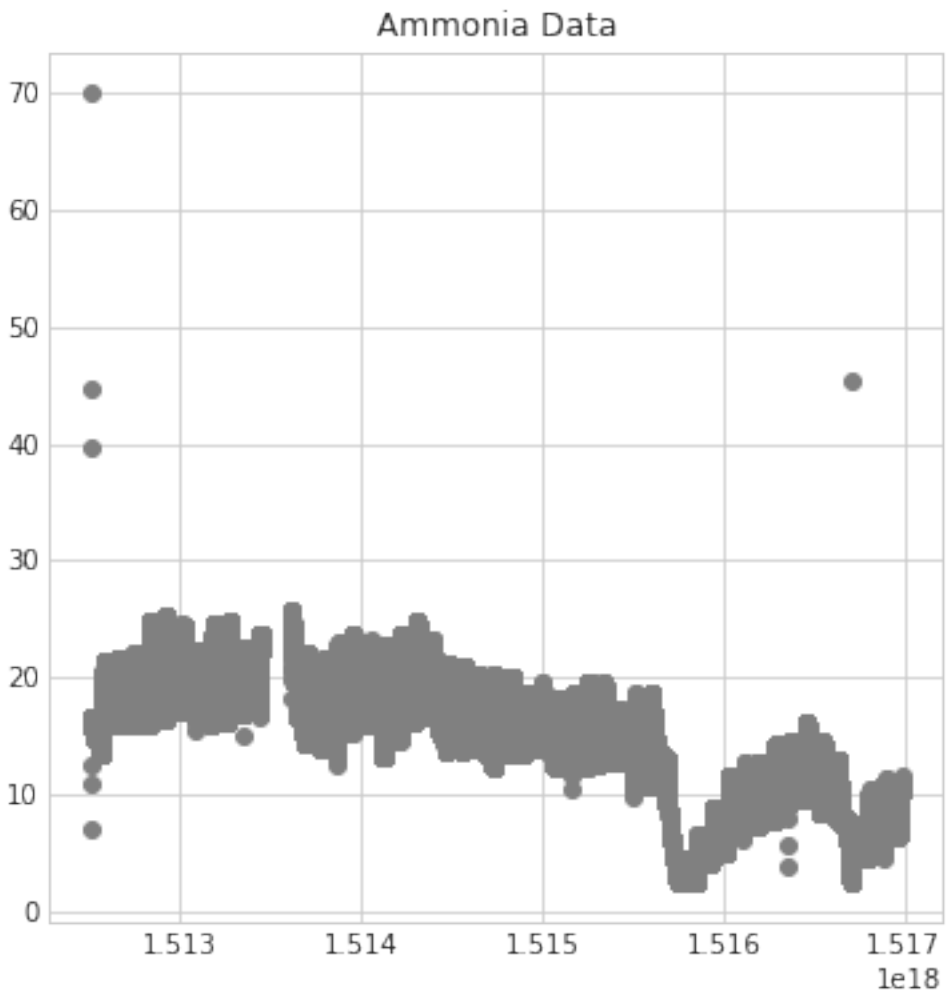
Set values of axis y for plotting.

```
In [31]: y = df_amm['Value']
```

Setup training data.

```
In [32]: X = np.array([x,y]).T
```

```
In [33]: fig=plt.figure(figsize=(6,6))
         fig.add_subplot(111)
         plt.plot(X[:,0],X[:,1],'o', color=[0.5,0.5,0.5])
         #plt.xlim([-5,30])
         #plt.ylim([-3.,3.])
         plt.title('Ammonia Data')
         plt.show()
```



Two sets of forests are trained, F0 and F1. F0 is the standard IF, which corresponds to extension level 0 in the context of EIF. F1 is the EIF with extension 1, which in the case of 2_D data (as in here), is the fully extended case.

```
In [34]: %%time
         F0  = iso.iForest(X,ntrees=100, sample_size=256, ExtensionLevel=0)

CPU times: user 730 ms, sys: 0 ns, total: 730 ms
Wall time: 825 ms
```

```
In [35]: %%time
         F1  = iso.iForest(X,ntrees=100, sample_size=256, ExtensionLevel=1)

CPU times: user 840 ms, sys: 0 ns, total: 840 ms
Wall time: 911 ms
```

Function compute_paths returns that anomaly score. It computes the depth each points reaches in each trained tree, and converts the ensemble aggregate to an anomaly score.

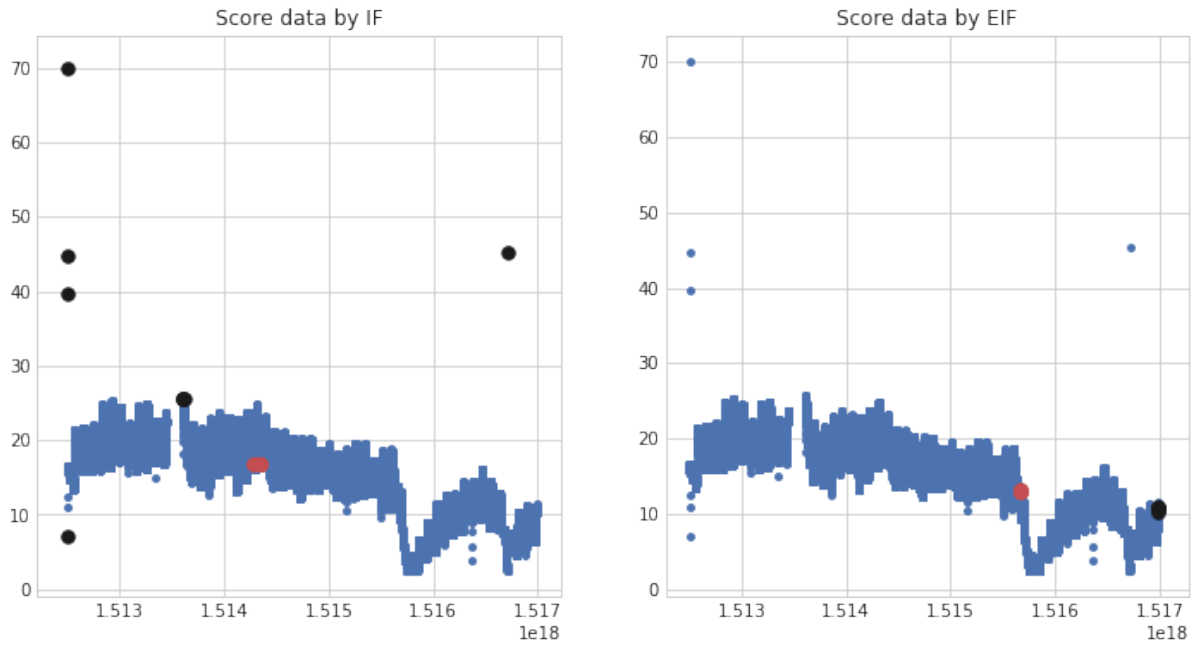
```
In [36]: %%time
         # Score the training data itself to see the distribution of
         # the anomaly scores each point receives.
         S0 = F0.compute_paths(X_in=X)
```

```
CPU times: user 2min 32s, sys: 20.2 ms, total: 2min 32s
Wall time: 2min 34s
```

```
In [37]: %%time
         S1 = F1.compute_paths(X_in=X)
```

```
CPU times: user 2min 33s, sys: 12.1 ms, total: 2min 33s
Wall time: 2min 35s
```

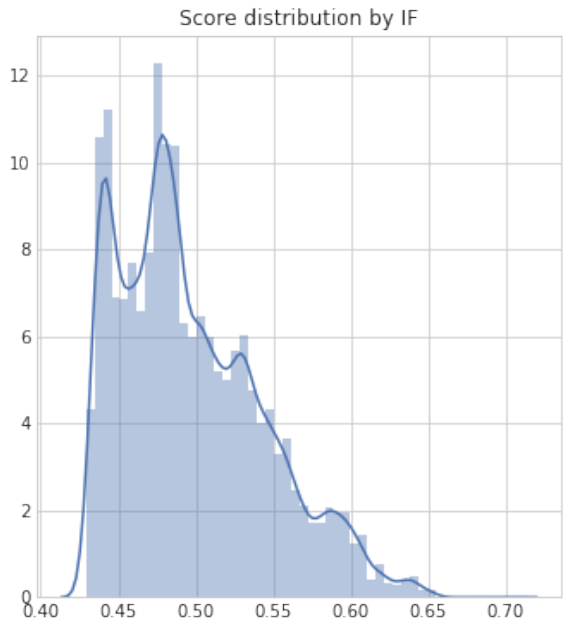
```
In [38]: ss0=np.argsort(S0)
         ss1=np.argsort(S1)

         f = plt.figure(figsize=(12,6))
         plt.subplot(1,2,1)
         plt.scatter(x,y,s=15,c='b',edgecolor='b')
         plt.scatter(x[ss0[-10:]],y[ss0[-10:]],s=55,c='k')
         plt.scatter(x[ss0[:10]],y[ss0[:10]],s=55,c='r')
         plt.title('Score data by IF')

         plt.subplot(1,2,2)
         plt.scatter(x,y,s=15,c='b',edgecolor='b')
         plt.scatter(x[ss1[-10:]],y[ss1[-10:]],s=55,c='k')
         plt.scatter(x[ss1[:10]],y[ss1[:10]],s=55,c='r')
         plt.title('Score data by EIF')
         plt.show()
```

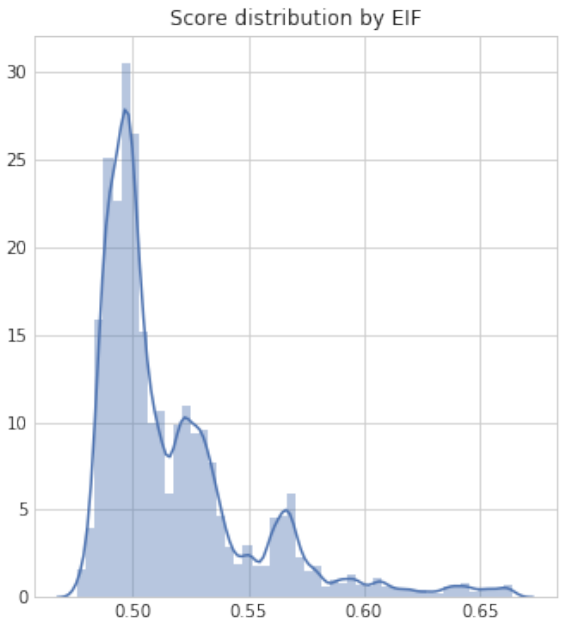


The distribution of the anomaly scores is shown above. By definition, anomalies are those that occur less frequently. So it makes sense that the number of points with higher anomaly scores reduces as the score increases.

The black dots in the above plotting clearly shows that the four point anomalies are missed by EIF.

```
In [39]: f = plt.figure(figsize=(12,6))
         plt.subplot(1,2,1)
         sb.distplot(S0, kde=True, color="b")
         plt.title('Score distribution by IF')

         plt.subplot(1,2,2)
         sb.distplot(S1, kde=True, color="b")
         plt.title('Score distribution by EIF')
         plt.show()
```





The following code is used to get a meaningful threshold to distinguish anomalies and normal data.

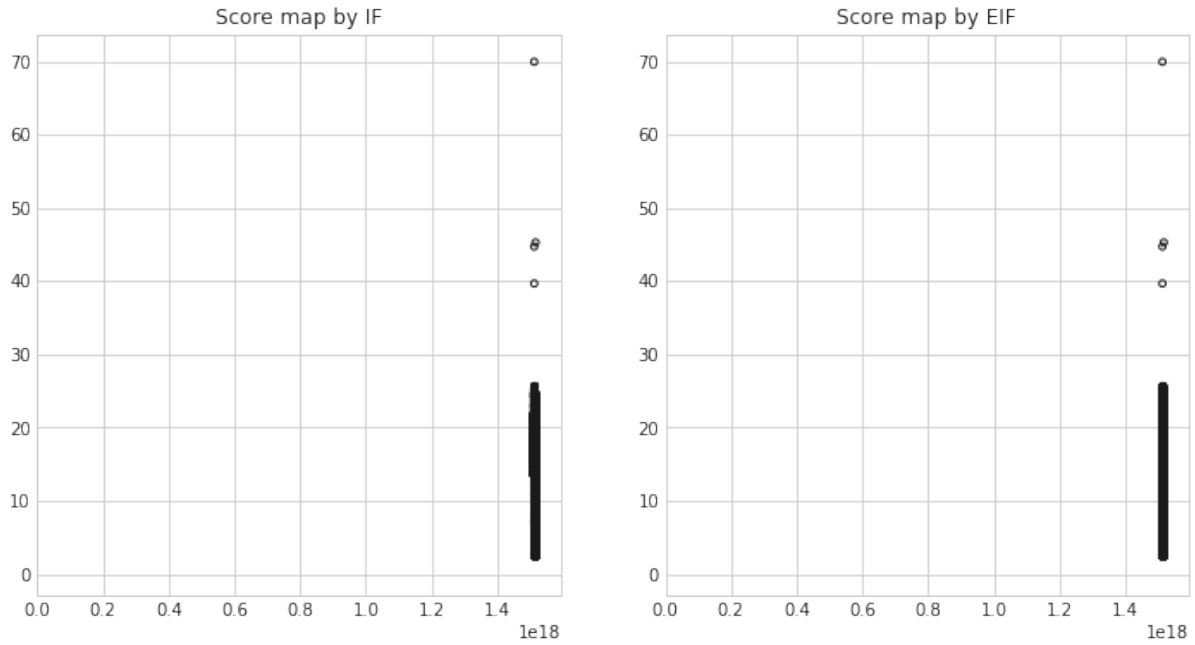
```
In [40]: anomalies0 = S0[S0 > 0.7]

In [41]: anomalies0.size

Out[41]: 1

In [42]: anomalies1 = S1[S1 > 0.7]

In [43]: anomalies1.size

Out[43]: 0

In [44]: anomalies0 = S0[S0 > 0.65]

In [45]: anomalies0.size

Out[45]: 157

In [46]: anomalies1 = S1[S1 > 0.65]

In [47]: anomalies1.size

Out[47]: 356
```

A meshgrid is created on a square domain. Each point on the grid is then scored using the trained forests. The resulting score map is visualized using contour plots.

```
In [48]: xx, yy = np.meshgrid(np.linspace(-5, 30, 30), np.linspace(-3, 3, 30))
```

```
        S0 = F0.compute_paths(X_in=np.c_[xx.ravel(), yy.ravel()])
        S0 = S0.reshape(xx.shape)

        S1 = F1.compute_paths(X_in=np.c_[xx.ravel(), yy.ravel()])
        S1 = S1.reshape(xx.shape)
```

In [49]:
```
f = plt.figure(figsize=(12,6))

ax1 = f.add_subplot(121)
levels = np.linspace(np.min(S0),np.max(S0),10)
CS = ax1.contourf(xx, yy, S0, levels, cmap=plt.cm.YlOrRd)
plt.scatter(x,y,s=15,c='None',edgecolor='k')
plt.title('Score map by IF')

ax2 = f.add_subplot(122)
levels = np.linspace(np.min(S1),np.max(S0),10)
CS = ax2.contourf(xx, yy, S1, levels, cmap=plt.cm.YlOrRd)
plt.scatter(x,y,s=15,c='None',edgecolor='k')
plt.title('Score map by EIF')

plt.show()
```



In [50]:
```
anomalies = S0[S0 > 0.7]
```

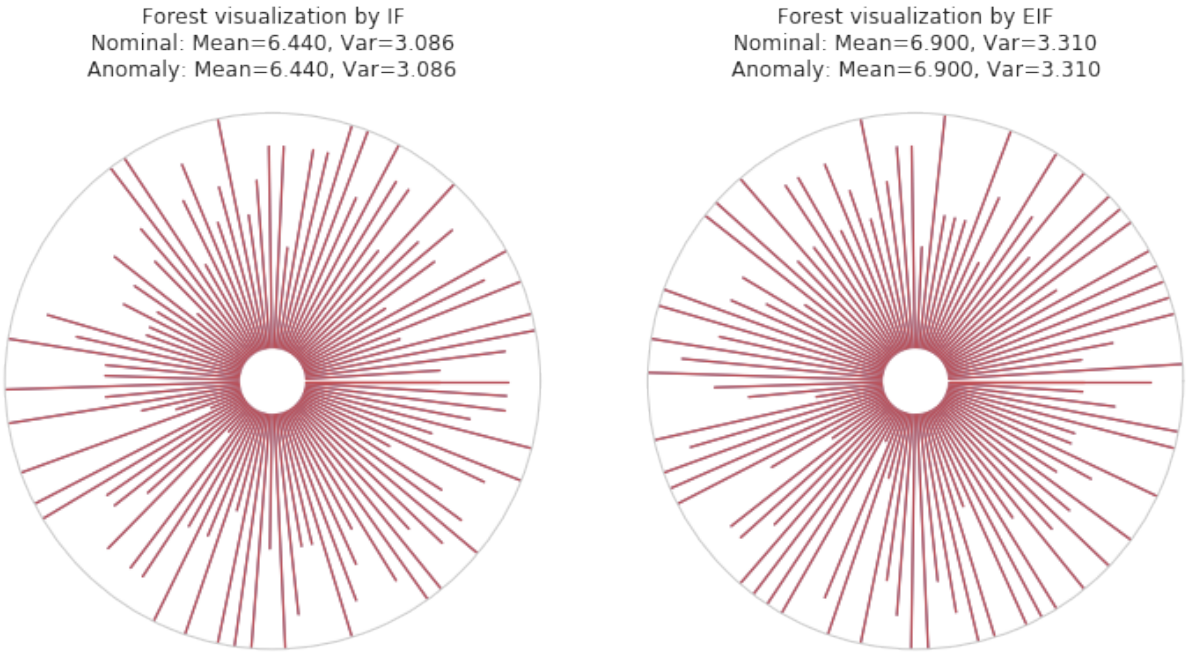
In [51]:
```
anomalies
```

```
Out[51]: array([], dtype=float64)

In [52]: anomalies = S0[S0 > 0.65]

In [53]: anomalies.size

Out[53]: 900
```

For each case, the forest is visualized by the same for temperature data.

```
In [54]: Sorted=False
         fig = plt.figure(figsize=(12,6))
         ax1 = plt.subplot(121, projection='polar')
         rn, thetan = getVals(F0,np.array([10,0]),sorted=Sorted)
         for j in range(len(rn)):
             ax1.plot([thetan[j],thetan[j]], [1,rn[j]], color='b',alpha=1,lw=1)

         ra, thetaa = getVals(F0,np.array([-5,-3]),sorted=Sorted)
         for j in range(len(ra)):
             ax1.plot([thetaa[j],thetaa[j]], [1,ra[j]], color='r',alpha=0.9,lw=1.3)

         title="Forest visualization by IF\nNominal: Mean={0:.3f}, Var={1:.3f}\n"+ \
         "Anomaly: Mean={2:.3f}, Var={3:.3f}"
         ax1.set_title(title.format(np.mean(rn),np.var(rn),np.mean(ra),np.var(ra)))

         ax1.set_xticklabels([])
         ax1.set_xlabel("Anomaly")
         ax1.set_ylim(0,F0.limit)

         ax1.axes.get_xaxis().set_visible(False)
         ax1.axes.get_yaxis().set_visible(False)
         #ax1.text(0,F0.limit+0.4,"800 Trees, full depth")

         ax2 = plt.subplot(122, projection='polar')
         rn, thetan = getVals(F1,np.array([10,0]),sorted=Sorted)
         for j in range(len(rn)):
             ax2.plot([thetan[j],thetan[j]], [1,rn[j]], color='b',alpha=1,lw=1)

         ra, thetaa = getVals(F1,np.array([-5,-3]),sorted=Sorted)
         for j in range(len(ra)):
             ax2.plot([thetaa[j],thetaa[j]], [1,ra[j]], color='r',alpha=0.9,lw=1.3)


         title=("Forest visualization by EIF\nNominal: Mean={0:.3f}, Var={1:.3f}\n" +
                "Anomaly: Mean={2:.3f}, Var={3:.3f}")
         ax2.set_title(title.format(np.mean(rn),np.var(rn),np.mean(ra),np.var(ra)))

         ax2.set_xticklabels([])
         ax2.set_xlabel("Anomaly")
```

```
ax2.set_ylim(0,F0.limit)

ax2.axes.get_xaxis().set_visible(False)
ax2.axes.get_yaxis().set_visible(False)
```



IF actually works better than Extended Isolation Forest on the test data.

Now, sklearn.ensemble.IsolationForest is used to detect anomalies in test data.

```
In [55]: import numpy as np
         import pandas as pd
         import matplotlib.pyplot as plt
         from sklearn.ensemble import IsolationForest
```

Read data from csv file.

```
In [56]: df_temp =  pd.read_csv("Temp.csv")
```

Change timestamp to epoch.

```
In [57]: df_temp['Timestamp'] = \
         pd.to_datetime(df_temp['Timestamp']).values.astype(np.int64)
```

Convert dataframe to array.

```
In [58]: temp_train= np.array(df_temp)
```

Make prediction of temperature data.

```
In [59]: %%time
         # fit the model
         clf = IsolationForest()
         clf.fit(temp_train)
         pred = clf.predict(temp_train)
```

```
CPU times: user 2.86 s, sys: 325 ms, total: 3.18 s
Wall time: 3.24 s
```

Merge predict result with test data.

```
In [60]: df_temp['anomaly']=pred
```

Get test result.

```
In [61]: outliers=df_temp.loc[df_temp['anomaly']==-1]
         outlier_index=list(outliers.index)
         #print(outlier_index)
         #Find the number of anomalies and normal points here points
         #classified -1 are anomalous
         print(df_temp['anomaly'].value_counts())
```

```
 1    70504
-1     7835
Name: anomaly, dtype: int64
```

Change epoch back to time.

```
In [62]: df_temp['Timestamp'] = \
         pd.to_datetime(df_temp['Timestamp']).values.astype(np.datetime64)
```

Display result.

```
In [63]: df_temp
```

Out[63]:

	Timestamp	Value	anomaly
0	2017-11-01 14:20:00	18.144899	-1
1	2017-11-01 14:21:00	18.148600	-1
2	2017-11-01 14:22:00	18.145201	-1
3	2017-11-01 14:23:00	18.164000	-1
4	2017-11-01 14:24:00	18.167101	-1
5	2017-11-01 14:25:00	18.166000	-1
6	2017-11-01 14:26:00	18.171200	-1
7	2017-11-01 14:27:00	18.176701	-1
8	2017-11-01 14:28:00	18.184601	-1
9	2017-11-01 14:29:00	18.174200	-1
10	2017-11-01 14:30:00	18.167101	-1
11	2017-11-01 14:31:00	18.168800	-1
12	2017-11-01 14:32:00	18.181801	-1
13	2017-11-01 14:33:00	18.175900	-1

14	2017-11-01 14:34:00	18.176901	-1
15	2017-11-01 14:35:00	18.176701	-1
16	2017-11-01 14:36:00	18.178101	-1
17	2017-11-01 14:37:00	18.175301	-1
18	2017-11-01 14:38:00	18.176701	-1
19	2017-11-01 14:39:00	18.173599	-1
20	2017-11-01 14:40:00	18.198200	-1
21	2017-11-01 14:41:00	18.193100	-1
22	2017-11-01 14:42:00	18.185900	-1
23	2017-11-01 14:43:00	18.197800	-1
24	2017-11-01 14:44:00	18.179701	-1
25	2017-11-01 14:45:00	18.181101	-1
26	2017-11-01 14:46:00	18.182199	-1
27	2017-11-01 14:47:00	18.185499	-1
28	2017-11-01 14:48:00	18.184900	-1
29	2017-11-01 14:49:00	18.185900	-1
...	...	...	...
78309	2018-01-09 19:26:00	11.914600	-1
78310	2018-01-09 19:27:00	11.908800	-1
78311	2018-01-09 19:28:00	11.919100	-1
78312	2018-01-09 19:29:00	11.925000	-1
78313	2018-01-09 19:30:00	11.937900	-1
78314	2018-01-09 19:31:00	11.943800	-1
78315	2018-01-09 19:32:00	11.945400	-1
78316	2018-01-09 19:33:00	11.951000	-1
78317	2018-01-09 19:34:00	11.961400	-1
78318	2018-01-09 19:35:00	11.968400	-1
78319	2018-01-09 19:36:00	11.972800	-1
78320	2018-01-09 19:37:00	11.978900	-1
78321	2018-01-09 19:38:00	11.988800	-1
78322	2018-01-09 19:39:00	11.981600	-1
78323	2018-01-09 19:40:00	11.968700	-1
78324	2018-01-09 19:41:00	11.964000	-1
78325	2018-01-09 19:42:00	11.955300	-1
78326	2018-01-09 19:43:00	11.958300	-1
78327	2018-01-09 19:44:00	11.945200	-1
78328	2018-01-09 19:45:00	11.948100	-1
78329	2018-01-09 19:46:00	11.962700	-1
78330	2018-01-09 19:47:00	11.949700	-1
78331	2018-01-09 19:48:00	11.942300	-1
78332	2018-01-09 19:49:00	11.936400	-1
78333	2018-01-09 19:50:00	11.933400	-1
78334	2018-01-09 19:51:00	11.936600	-1
78335	2018-01-09 19:52:00	11.940700	-1
78336	2018-01-09 19:53:00	11.949600	-1
78337	2018-01-09 19:54:00	11.965600	-1

```
        78338 2018-01-09 19:55:00  11.980300       -1

        [78339 rows x 3 columns]
```

Read ammonia data from csv file.

```
In [64]: df_amm =  pd.read_csv("Ammonia.csv")
```

Change timestamp to epoch.

```
In [65]: df_amm['Timestamp'] = \
         pd.to_datetime(df_amm['Timestamp']).values.astype(np.int64)
```

Convert dataframe to array.

```
In [66]: amm_train= np.array(df_amm)
```

Make prediction of ammonia data set.

```
In [67]: %%time
         # fit the model
         clf = IsolationForest()
         clf.fit(amm_train)
         pred = clf.predict(amm_train)
```

```
CPU times: user 2.66 s, sys: 234 ms, total: 2.89 s
Wall time: 2.98 s
```

Merge prediction score with data set.

```
In [68]: df_amm['anomaly']=pred
```

Get result.

```
In [69]: outliers=df_amm.loc[df_amm['anomaly']==-1]
         outlier_index=list(outliers.index)
         #print(outlier_index)
         #Find the number of anomalies and normal points here points
         #classified -1 are anomalous
         print(df_amm['anomaly'].value_counts())
```

```
 1    64731
-1     7195
Name: anomaly, dtype: int64
```

Convert epoch back to timestamp.

```
In [70]: df_amm['Timestamp'] = \
         pd.to_datetime(df_amm['Timestamp']).values.astype(np.datetime64)
```

Display result.

```
In [71]: df_amm
```

Out[71]:

	Timestamp	Value	anomaly
0	2017-12-05 19:00:00	16.245501	-1
1	2017-12-05 19:01:00	16.227800	-1
2	2017-12-05 19:02:00	16.292000	-1
3	2017-12-05 19:03:00	15.719300	-1
4	2017-12-05 19:04:00	15.761400	-1
5	2017-12-05 19:05:00	15.847900	-1
6	2017-12-05 19:06:00	15.800900	-1
7	2017-12-05 19:07:00	15.801100	-1
8	2017-12-05 19:08:00	15.973500	-1
9	2017-12-05 19:09:00	16.062700	-1
10	2017-12-05 19:10:00	16.108700	-1
11	2017-12-05 19:11:00	15.975700	-1
12	2017-12-05 19:12:00	16.643101	1
13	2017-12-05 19:13:00	16.648600	1
14	2017-12-05 19:14:00	16.607901	1
15	2017-12-05 19:15:00	16.652300	1
16	2017-12-05 19:16:00	16.562700	-1
17	2017-12-05 19:17:00	16.515800	-1
18	2017-12-05 19:18:00	16.475201	-1
19	2017-12-05 19:19:00	16.557100	-1
20	2017-12-05 19:38:00	16.495199	-1
21	2017-12-05 19:39:00	16.452200	-1
22	2017-12-05 19:40:00	16.494600	-1
23	2017-12-05 19:41:00	16.453400	-1
24	2017-12-05 19:42:00	39.700600	-1
25	2017-12-05 19:43:00	44.756802	-1
26	2017-12-05 19:44:00	69.996399	-1
27	2017-12-05 19:45:00	7.129600	-1
28	2017-12-05 19:46:00	11.029300	-1
29	2017-12-05 19:47:00	12.425300	-1
...	...	...	...
71896	2018-01-26 17:39:00	10.423400	-1
71897	2018-01-26 17:40:00	10.451100	-1
71898	2018-01-26 17:41:00	10.691800	-1
71899	2018-01-26 17:42:00	10.617000	-1
71900	2018-01-26 17:43:00	10.616700	-1
71901	2018-01-26 17:44:00	10.507100	-1
71902	2018-01-26 17:45:00	10.443300	-1
71903	2018-01-26 17:46:00	10.381100	-1
71904	2018-01-26 17:47:00	10.345200	-1
71905	2018-01-26 17:48:00	10.275300	-1
71906	2018-01-26 17:49:00	10.311600	-1
71907	2018-01-26 17:50:00	10.531100	-1
71908	2018-01-26 17:51:00	10.531300	-1
71909	2018-01-26 17:52:00	10.458400	-1

```
71910 2018-01-26 17:53:00  10.466800       -1
71911 2018-01-26 17:54:00  10.466800       -1
71912 2018-01-26 17:55:00  10.395200       -1
71913 2018-01-26 17:56:00  10.360300       -1
71914 2018-01-26 17:57:00  10.324500       -1
71915 2018-01-26 17:58:00  10.431200       -1
71916 2018-01-26 17:59:00  10.504600       -1
71917 2018-01-26 18:00:00  10.324700       -1
71918 2018-01-26 18:01:00  10.420600       -1
71919 2018-01-26 18:02:00  10.528200       -1
71920 2018-01-26 18:03:00  10.455500       -1
71921 2018-01-26 18:04:00  10.281000       -1
71922 2018-01-26 18:05:00  10.424000       -1
71923 2018-01-26 18:06:00  10.457700       -1
71924 2018-01-26 18:07:00  10.422400       -1
71925 2018-01-26 18:08:00  10.352900       -1

[71926 rows x 3 columns]
```

The outlier-ness of IF is based on the score. Over 7000 points for both temperature and ammonia marked as anomalies above 0.6 as suggested by the author. In this case it is very difficult to find a feasible threshold for improvement. For instance, most of the anomalies are considered as inliers if 0.7 is set as threshold, but hundreds of normal points are still considered as anomalies if 0.65 is set as threshold. However, this algorithm might be used for cross-validation if the anomaly found by other algorithm get a higher score (like 0.65) using IF, then this point can be safely classified as an anomaly.

Robust Random Cut Forest

Use robust random cut forest to detect anomalies in test data.

```
In [1]: import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt
        import rrcf
        import pandas as pd
        import matplotlib.pyplot as plt
        import seaborn as sns
```

Read data from csv file.

```
In [2]: df_temp =  pd.read_csv("Temp.csv")
```

Convert dataframe to array.

```
In [3]: temp_train= np.array(df_temp['Value'])
```

Make prediction of test data.

```
In [4]: # Set tree parameters
        num_trees = 40
        shingle_size = 1
        tree_size = 256
```

Merge prediction result with test data.

```
In [5]: # Create a forest of empty trees
        forest = []
        for _ in range(num_trees):
            tree = rrcf.RCTree()
            forest.append(tree)
```

```
In [6]: # Use the "shingle" generator to create rolling window
        points = rrcf.shingle(temp_train, size=shingle_size)
```

```
In [7]: # Create a dict to store anomaly score of each point
        avg_codisp = {}
```

Display prediction result.

```
In [8]: %%time
        for index, point in enumerate(points):
            # For each tree in the forest...
            for tree in forest:
                # If tree is above permitted size, drop the oldest point
                # (FIFO)
                if len(tree.leaves) > tree_size:
                    tree.forget_point(index - tree_size)
                # Insert the new point into the tree
                tree.insert_point(point, index=index)
                # Compute codisp on the new point and take the average among
                # all trees
                if not index in avg_codisp:
                    avg_codisp[index] = 0
                avg_codisp[index] += tree.codisp(index) / num_trees
```

```
CPU times: user 11min 31s, sys: 185 ms, total: 11min 31s
Wall time: 11min 40s
```

This method takes almost 12 minutes for detection which is too long compared to other algorithms like LOF which only takes hundreds of milliseconds.

Convert score dictionary to data frame.

```
In [9]: temp_result = pd.DataFrame.from_dict(avg_codisp,'index')
```

Merge score with data.

```
In [10]: df_temp['Score'] = temp_result
```

Try different values of threshold to get anomalies.

```
In [11]: anomalies = df_temp.loc[df_temp['Score']>120]

In [12]: len(anomalies)

Out[12]: 30

In [13]: anomalies
```

Out[13]:

	Timestamp	Value	Score
6326	2017/11/7 12:38	16.982100	240.775000
6327	2017/11/7 12:39	16.974199	129.912500
8081	2017/11/10 14:37	14.233200	225.926667
8082	2017/11/10 14:38	10.074700	224.762500
8139	2017/11/10 15:40	15.121400	148.690104
8140	2017/11/10 15:41	12.731000	127.029167
8144	2017/11/10 15:45	0.000000	152.769673
17989	2017/11/17 12:08	0.000000	252.975000
17990	2017/11/17 12:09	0.000000	126.012500
17991	2017/11/17 12:10	15.785600	150.755547
23756	2017/11/21 12:26	0.000000	252.225000
23757	2017/11/21 12:27	0.000000	125.637500
23761	2017/11/21 12:31	15.368100	121.414583
24112	2017/11/24 14:19	0.000000	245.115179
24113	2017/11/24 14:20	0.000000	123.090179
24118	2017/11/24 14:25	15.363900	165.154968
28686	2017/11/30 13:13	0.000000	250.500000
30309	2017/12/5 12:54	14.546800	171.106964
30594	2017/12/5 17:39	0.000000	243.250000
30602	2017/12/5 17:47	0.000000	120.887500
30654	2017/12/5 19:42	13.076000	204.052083
30655	2017/12/5 19:43	12.322200	145.385417
33778	2017/12/7 23:46	14.668300	162.724122
40360	2017/12/12 13:50	13.636600	124.976042
44997	2017/12/15 19:22	13.527400	132.051732
48515	2017/12/20 1:33	14.009000	132.680690
48516	2017/12/20 1:34	14.024500	135.863333
77245	2018/1/9 1:37	11.976000	125.731692
77947	2018/1/9 13:19	9.805300	208.825000
77948	2018/1/9 13:20	9.549650	122.287500

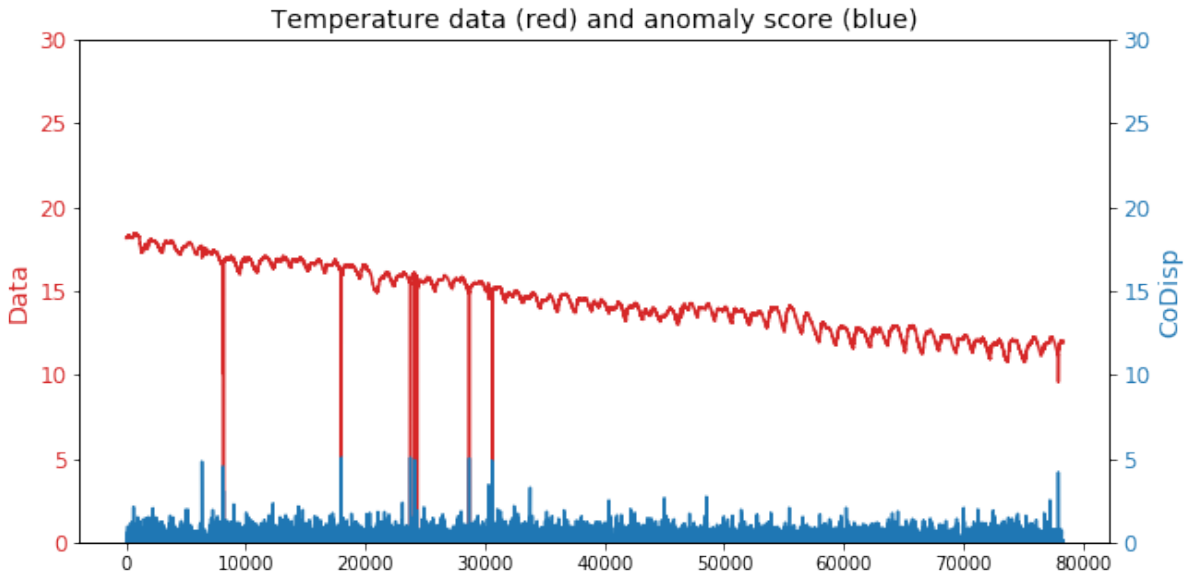
The top 30 records with highest scores were listed above as it is not easy to find a feasible threshold to split the anomalies. The 6 point anomalies (value=0) detected by other algorithms are missing from this list. This list also contains some point which may be considered as noise. For example, for record [2017/11/7 12:38, 16.982100], its value only is less than 0.7 different from its closest point but the score is 240; and for record [2018/1/9 1:37 11.976000], its value only is less than 0.08 different from its closest point but the score is 125.

Scale scores for plotting.

```
In [14]: for index in range(len(avg_codisp)):
             avg_codisp[index]=avg_codisp[index]/50
```

```
In [15]: fig, ax1 = plt.subplots(figsize=(10, 5))

         color = 'tab:red'
         ax1.set_ylabel('Data', color=color, size=14)
         ax1.plot(temp_train, color=color)
         ax1.tick_params(axis='y', labelcolor=color, labelsize=12)
         ax1.set_ylim(0,30)
         ax2 = ax1.twinx()
         color = 'tab:blue'
         ax2.set_ylabel('CoDisp', color=color, size=14)
         ax2.plot(pd.Series(avg_codisp).sort_index(), color=color)
         ax2.tick_params(axis='y', labelcolor=color, labelsize=12)
         ax2.grid('off')
         ax2.set_ylim(0, 30)
         plt.title('Temperature data (red) and anomaly score (blue)', size=14)
```

```
Out[15]: Text(0.5,1,'Temperature data (red) and anomaly score (blue)')
```



Test on ammonia data.

Read data from csv file.

```
In [16]: df_amm =  pd.read_csv("Ammonia.csv")
```

Convert dataframe to array.

```
In [17]: amm_train= np.array(df_amm['Value'])
```

Make prediction of test data.

```
In [18]: # Set tree parameters
         num_trees = 40
         shingle_size = 1
         tree_size = 256
```

Merge prediction result with test data.

```
In [19]: # Create a forest of empty trees
         forest = []
         for _ in range(num_trees):
             tree = rrcf.RCTree()
             forest.append(tree)
```

Get test result.

```
In [20]: # Use the "shingle" generator to create rolling window
         points = rrcf.shingle(amm_train, size=shingle_size)
```

There 7841 anomalies with score -1.

Change epoch back to time.

```
In [21]: # Create a dict to store anomaly score of each point
         avg_codisp = {}
```

Display result.

```
In [22]: %%time
         for index, point in enumerate(points):
             # For each tree in the forest...
             for tree in forest:
                 # If tree is above permitted size, drop the oldest point
                 # (FIFO)
                 if len(tree.leaves) > tree_size:
                     tree.forget_point(index - tree_size)
                 # Insert the new point into the tree
                 tree.insert_point(point, index=index)
                 # Compute codisp on the new point and take the average among
                 # all trees
                 if not index in avg_codisp:
                     avg_codisp[index] = 0
                 avg_codisp[index] += tree.codisp(index) / num_trees
```

```
CPU times: user 15min 3s, sys: 127 ms, total: 15min 4s
Wall time: 15min 19s
```

This method takes over 15 minutes for detection which is too long compared to other algorithms like LOF which only takes hundreds of milliseconds. Convert score dictionary to data frame.

```
In [23]: amm_result = pd.DataFrame.from_dict(avg_codisp,'index')
```

Merge score with data.

```
In [24]: df_amm['Score'] = amm_result
```

Try different values of threshold to get anomalies.

```
In [25]: anomalies = df_amm.loc[df_amm['Score']>130]

In [26]: len(anomalies)

Out[26]: 17

In [27]: anomalies
```

Out[27]:

	Timestamp	Value	Score
9729	2017-12-12 13:49:00	15.552900	206.675000
11190	2017-12-13 14:10:00	18.264601	152.091667
11191	2017-12-13 14:11:00	18.697500	138.970833
12510	2017-12-14 12:16:00	16.183201	164.291667
14123	2017-12-15 15:18:00	14.998800	197.975000
21245	2017-12-22 9:41:00	16.111099	151.334077
21260	2017-12-22 9:56:00	15.994200	136.693408
41609	2018-01-05 14:11:00	10.544300	227.675000
47319	2018-01-09 13:26:00	9.701480	143.184509
57386	2018-01-16 13:36:00	6.052440	163.979924
57389	2018-01-16 13:39:00	8.159490	146.819097
61591	2018-01-19 12:17:00	5.649060	217.175000
61592	2018-01-19 12:18:00	3.840480	148.241667
67407	2018-01-23 14:30:00	45.311798	224.180952
69564	2018-01-25 2:27:00	7.579580	133.757652
70192	2018-01-25 12:55:00	4.522930	138.131250
71245	2018-01-26 6:28:00	9.218790	134.025000

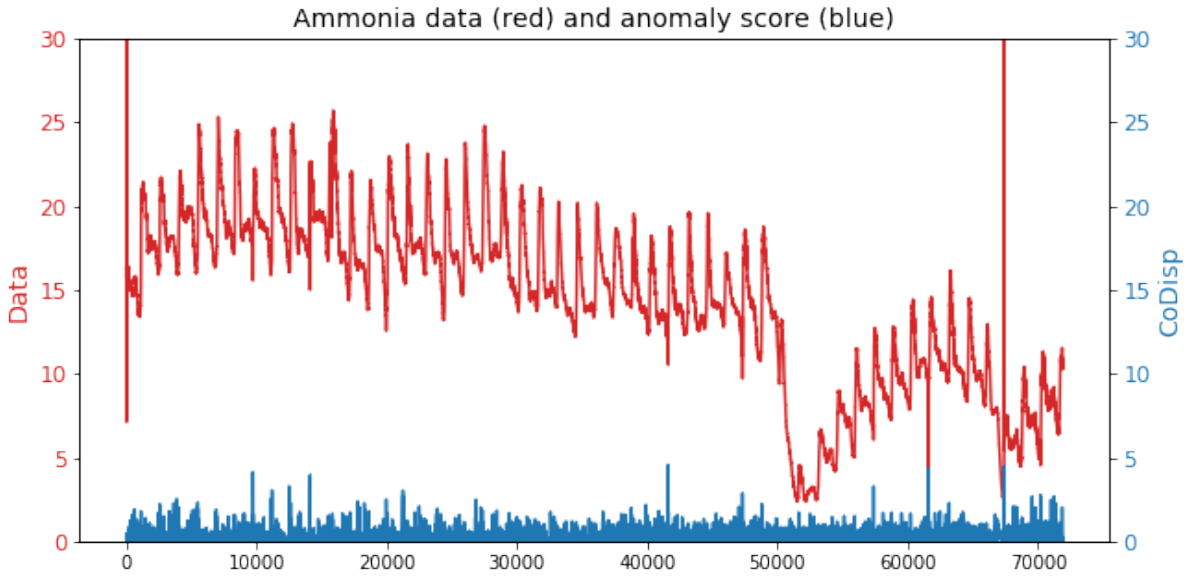
The top 10 records with highest scores were listed above as it is not easy to find a feasible threshold to split the anomalies. The 3 point anomalies detected by other algorithms are missing from this list. This list also contains some point which may be considered as noise. For example, for record [2017-12-15 15:18:00 14.998800], its value only is less than 0.1 different from its closest point but the score is 197; and for record [2017-12-14 12:16:00 16.183201], its value only is less than 0.05 different from its closest points but the score is 164.

Scale scores for plotting.

```
In [28]: for index in range(len(avg_codisp)):
             avg_codisp[index]=avg_codisp[index]/50

In [29]: fig, ax1 = plt.subplots(figsize=(10, 5))

         color = 'tab:red'
         ax1.set_ylabel('Data', color=color, size=14)
         ax1.plot(amm_train, color=color)
         ax1.tick_params(axis='y', labelcolor=color, labelsize=12)
         ax1.set_ylim(0,30)
```

```
ax2 = ax1.twinx()
color = 'tab:blue'
ax2.set_ylabel('CoDisp', color=color, size=14)
ax2.plot(pd.Series(avg_codisp).sort_index(), color=color)
ax2.tick_params(axis='y', labelcolor=color, labelsize=12)
ax2.grid('off')
ax2.set_ylim(0, 30)
plt.title('Ammonia data (red) and anomaly score (blue)', size=14)
```

```
Out[29]: Text(0.5,1,'Ammonia data (red) and anomaly score (blue)')
```



In summary, the execution time is too long which is expected as it takes time to construct trees and compute the depth and length. Some outliers are marked with low anomaly scores (like the zeros) whereas some normal points are marked with high anomaly scores which makes it difficult to find a good threshold to filter out anomalies.

3.4 Evaluation using Jupyter Notebook II

3.4.1 Test Data Sets

To evaluate the performance of each algorithm, four data sets from the RSM30 water monitoring system are used in this experiment. These data sets were provided by Primodal System and were retrieved from the Dundas wastewater treatment plant.

Table 3.3 summarizes the information of these four data sets.

Parameters	Number of Records	Date Time range
Temperature	395,715	2018/2/1 0:00:00 - 2018/11/8 14:58:00
Ammonia	395,715	2018/2/1 0:00:00 - 2018/11/8 14:58:00
Chloride	395,715	2018/2/1 0:00:00 - 2018/11/8 14:58:00
Potassium	395,715	2018/2/1 0:00:00 - 2018/11/8 14:58:00

TABLE 3.3: Water Quality Data

Figure 3.3 shows the test data set for water temperature from February 1, 2018 to November 8, 2018.

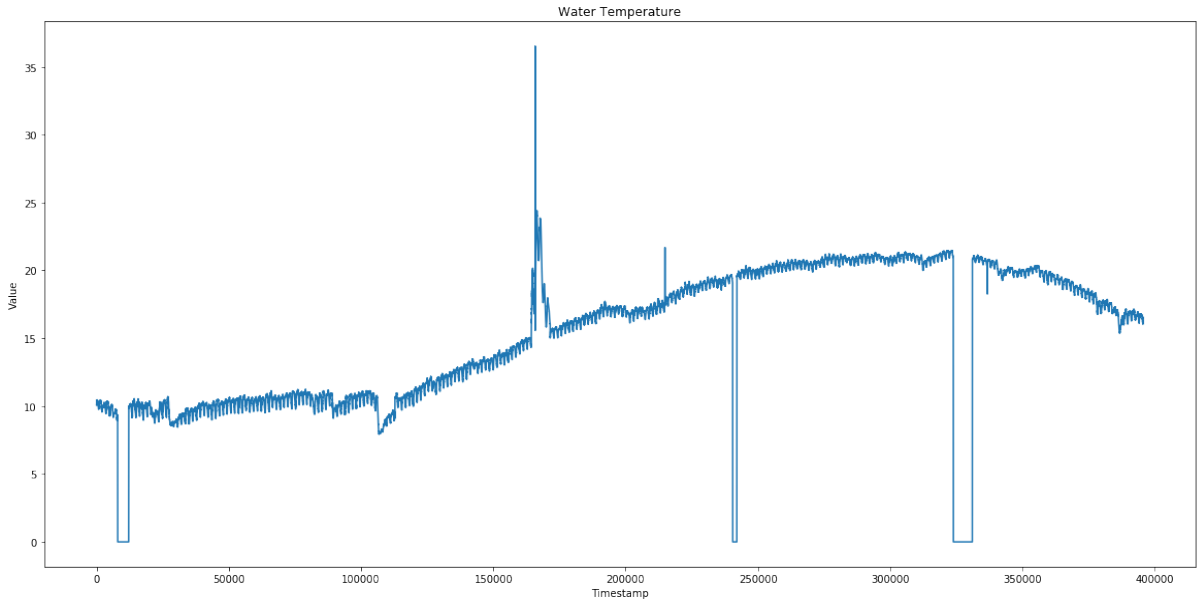


FIGURE 3.3: Temperature Data Set 2018

Figure 3.4 shows the test data set for ammonia data from February 1, 2018 to November 8, 2018.

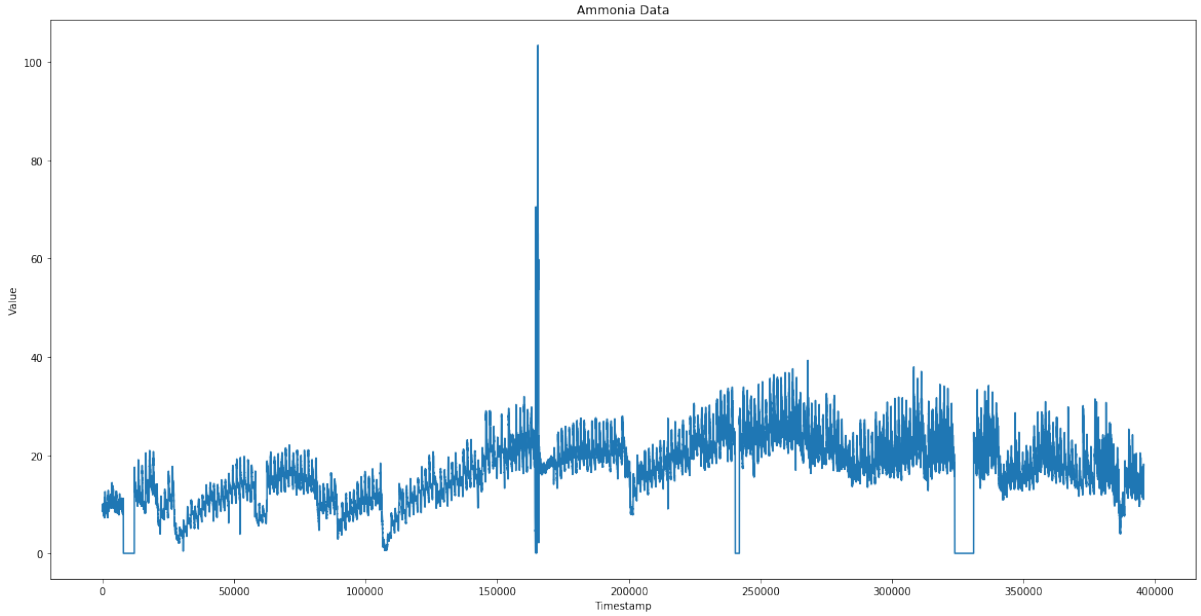


FIGURE 3.4: Ammonia Data Set 2018

Figure 3.5 shows the test data set for chloride data from February 1, 2018 to November 8, 2018.

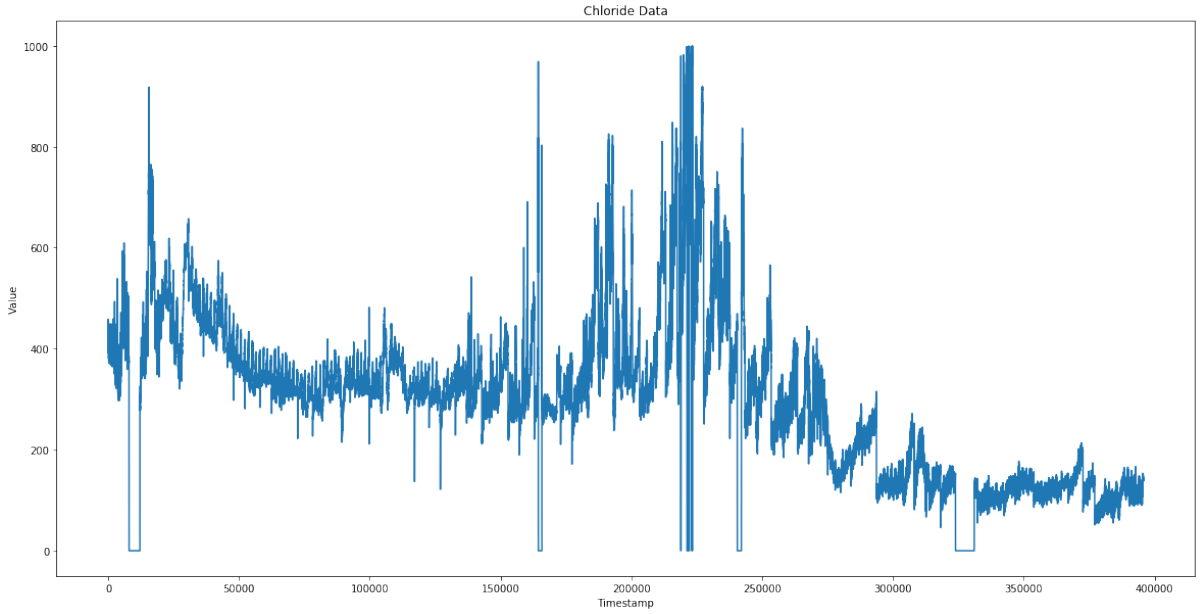


FIGURE 3.5: Chloride Data Set 2018

Figure 3.6 shows the test data set for potassium data from February 1, 2018 to November 8, 2018.

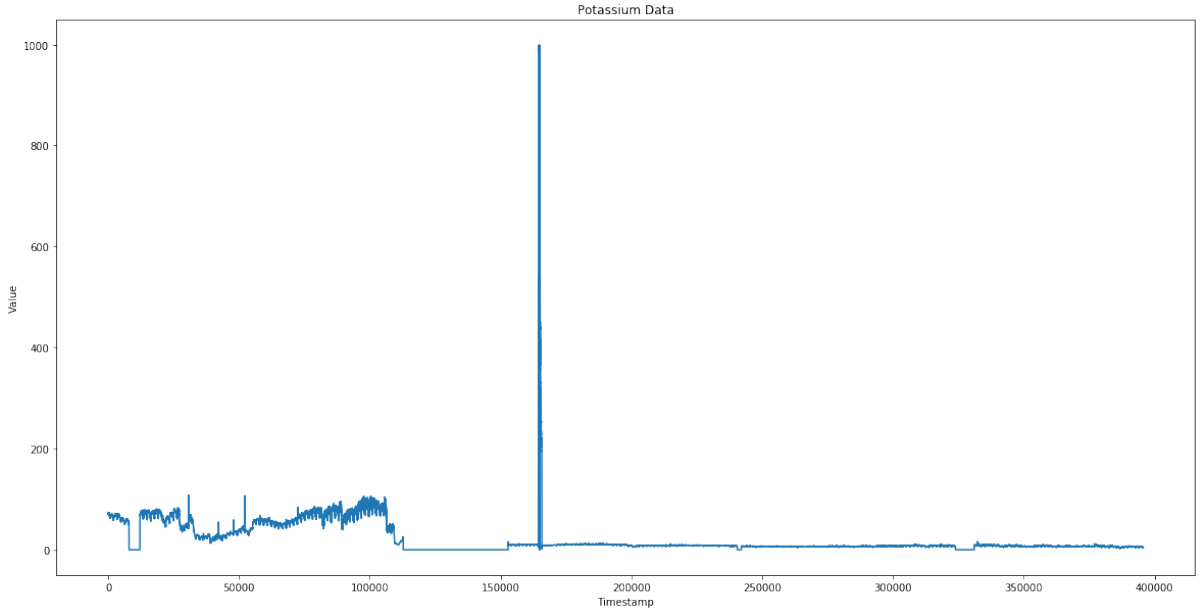


FIGURE 3.6: Potassium Data Set 2018

3.4.2 Notebook

Machine Learning

First of all, the unsupervised outlier detection class 'LocalOutlierFactor' in scikit-learn is used to to detect anomalies in temperature, ammonia, chloride and potassium data.

Run Python file for the auxiliary functions.

```
In [1]: %run anomaly_detection.py
```

At the beginning, functions are defined to get data from csv files, process and then display them.

Following methods are used to read test data from csv file.

```
In [2]: def get_temp_data():
            return pd.read_csv("2018temp.csv")

In [3]: def get_ammonia_data():
            return pd.read_csv("2018amm.csv")

In [4]: def get_chloride_data():
            return pd.read_csv("2018chloride.csv")

In [5]: def get_potassium_data():
            return pd.read_csv("2018potassium.csv")
```

Read temperature data from "2018temp.csv" file and save to temp. "2018temp.csv" contains two columns: "Timestamp" and "Value"

```
In [6]: temp = get_temp_data()
```

Display the general information of the test temperature data:

```
In [7]: print ("Data Range")
        print ("Start Date %s"%(temp.head(1)['Timestamp']))
        print ("End Date %s"%(temp.tail(1)['Timestamp']))
```

```
Data Range
Start Date 0    2/1/2018 0:00
Name: Timestamp, dtype: object
End Date 395713    11/8/2018 14:58
Name: Timestamp, dtype: object
```

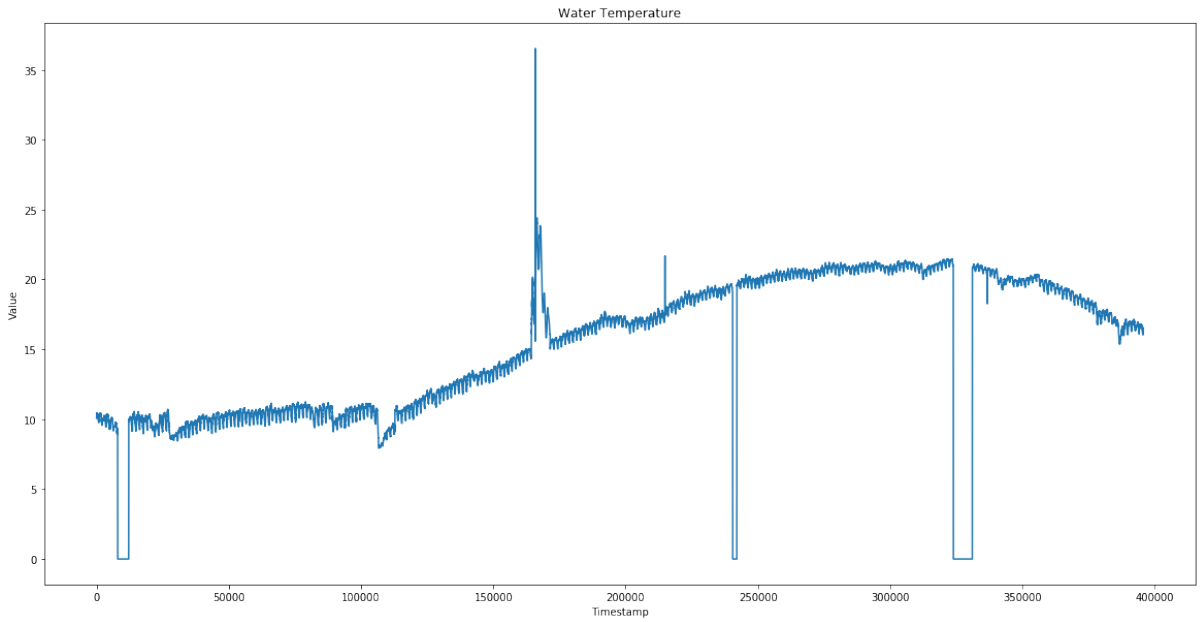
Get the total number of records in data set

```
In [8]: print ("Number of records: %s"%(temp.shape[0]))
```

```
Number of records: 395714
```

Now plot temperature test data.

```
In [9]: plot_original_data(temp, "Water Temperature")
```



The above output diagram clearly shows that there are several instances which are far from others.

Call method to retrieve ammonia test data from "2018amm.csv" file.

```
In [10]: ammonia = get_ammonia_data()
```

Display general information of ammonia test data.

```
In [11]: display_info(ammonia)
```

```
Data Range
Start Date 0    2/1/2018 0:00
Name: Timestamp, dtype: object
End Date 395713    11/8/2018 14:58
Name: Timestamp, dtype: object
num_values: 395714
```

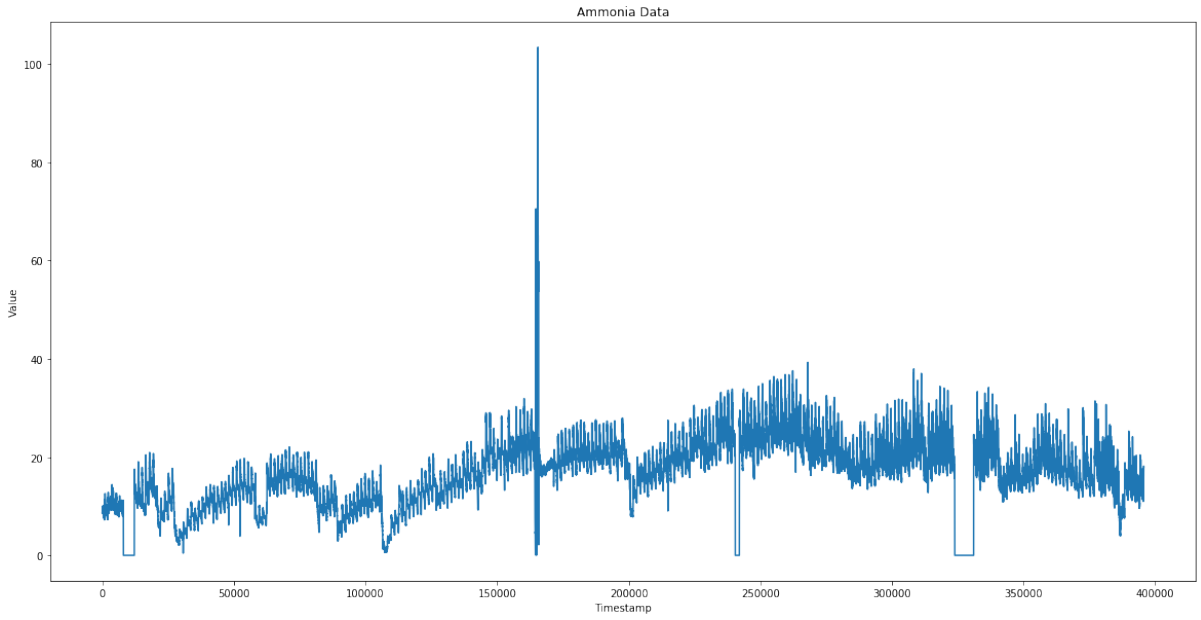
There are 395714 records in the ammonia data.

```
In [12]: print ("Number of records: %s"%(ammonia.shape[0]))
```

```
Number of records: 395714
```

Plot ammonia test data.

```
In [13]: plot_original_data(ammonia, "Ammonia Data")
```



Read chloride data from "2018chloride.csv" file.

```
In [14]: chloride = get_chloride_data()
```

Display the general data information.

```
In [15]: display_info(chloride)
```

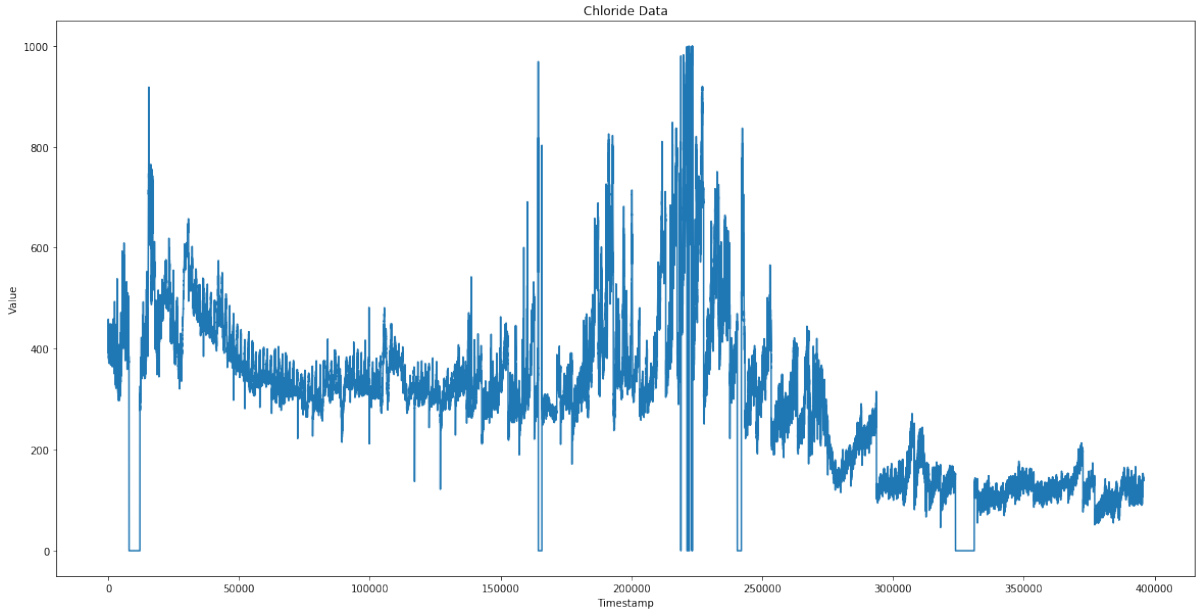
```
Data Range
Start Date 0    2/1/2018 0:00
Name: Timestamp, dtype: object
End Date 395713    11/8/2018 14:58
Name: Timestamp, dtype: object
num_values: 395714
```

```
In [16]: print ("Number of records: %s"%(chloride.shape[0]))
```

```
Number of records: 395714
```

Plot chloride data.

```
In [17]: plot_original_data(chloride, "Chloride Data")
```



Read potassium data from "2018potassium.csv" file.

```
In [18]: potassium =  get_potassium_data()
```

Display general data information.

```
In [19]: display_info(potassium)
```

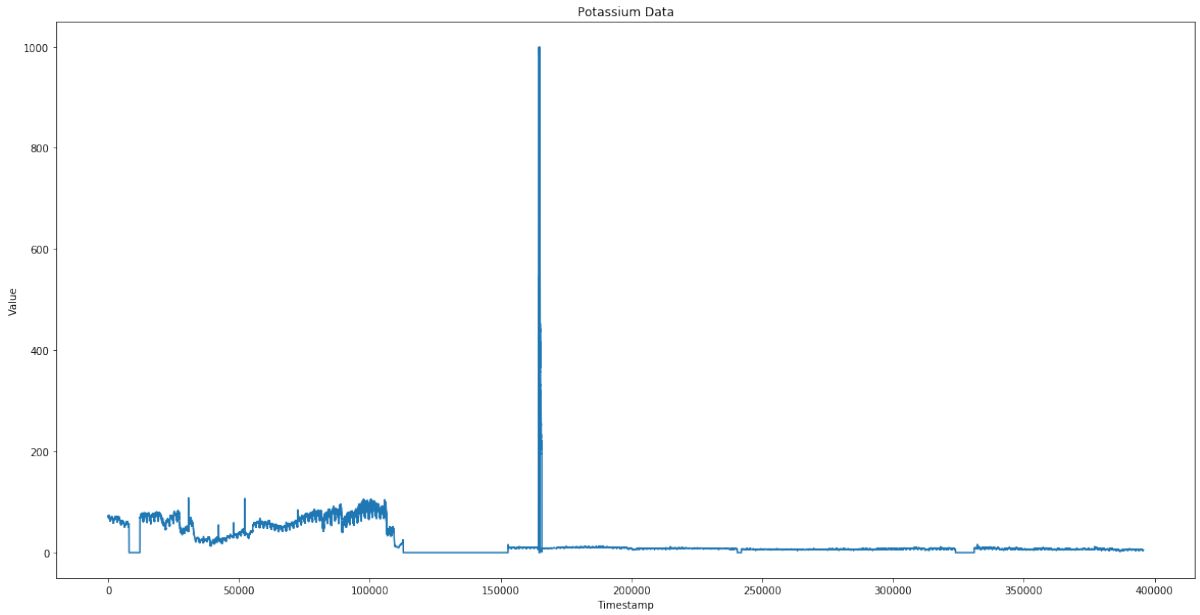
```
Data Range
Start Date 0    2/1/2018 0:00
Name: Timestamp, dtype: object
End Date 395713    11/8/2018 14:58
Name: Timestamp, dtype: object
```

```
num_values: 395714
```

```
In [20]: print ("Number of records: %s"%(potassium.shape[0]))
Number of records: 395714
```

Plot potassium data.

```
In [21]: plot_original_data(potassium, "Potassium Data")
```



Detect outlier using Local Outlier Factor which is from scikit-learn library.

Fist of all, detect anomalies in temperature data.

Function "process_data" creates index on "Timestamp" column for temperature data.

```
In [22]: i_temp = process_data(temp)
```

Get data of "Value" column which is used to fit LOF class later.

```
In [23]: temp_data = reshape_training_dataset(i_temp)
```

Initialize an instance of LocalOutlierFactor.

```
In [24]: clf = LocalOutlierFactor(metric='euclidean')
```

Get the prediction result.

```
In [25]: %%time
         temp_anomalies = lof_prediction(clf,temp_data, i_temp)
```

```
CPU times: user 5.16 s, sys: 396 ms, total: 5.55 s
Wall time: 5.81 s
```

```
In [26]: print ("Number of Anomalies:%s"%(temp_anomalies.shape[0]))
```

```
Number of Anomalies:156
```

Display all the anomaly instances with their score.

```
In [27]: print (temp_anomalies)
```

```
                        Value      isinlier
Timestamp
2018-04-16 09:59:00   7.997010 -1.500000e+07
2018-04-16 10:00:00   7.995570 -1.500000e+07
2018-04-16 10:01:00   7.998840 -1.500000e+07
2018-04-16 10:13:00   7.974030 -3.000000e+07
2018-04-16 10:14:00   7.972470 -3.000000e+07
2018-04-16 10:15:00   7.970760 -3.000000e+07
2018-04-16 10:16:00   7.972470 -3.000000e+07
2018-04-16 10:17:00   7.972600 -3.000000e+07
2018-04-16 10:18:00   7.971010 -3.000000e+07
2018-04-16 10:19:00   7.972600 -3.000000e+07
2018-04-16 11:03:00   7.944670 -3.000000e+07
2018-04-16 11:10:00   7.944760 -3.000000e+07
2018-04-16 11:11:00   7.943240 -3.000000e+07
2018-04-16 11:12:00   7.942960 -3.000000e+07
2018-04-16 11:13:00   7.944580 -3.000000e+07
2018-04-16 11:14:00   7.944760 -3.000000e+07
2018-04-16 11:15:00   7.944580 -3.000000e+07
2018-04-16 11:16:00   7.944760 -3.000000e+07
2018-04-16 11:18:00   7.943050 -3.000000e+07
2018-04-16 11:19:00   7.943240 -3.000000e+07
2018-04-16 11:20:00   7.943050 -3.000000e+07
2018-04-16 11:23:00   7.943150 -3.000000e+07
2018-04-16 11:24:00   7.942960 -3.000000e+07
2018-04-16 11:25:00   7.944670 -3.000000e+07
2018-04-16 11:42:00   7.966220 -3.000000e+07
2018-04-16 11:43:00   7.967860 -3.000000e+07
2018-04-16 11:44:00   7.971010 -3.000000e+07
2018-04-16 11:45:00   7.970950 -3.000000e+07
2018-04-16 11:46:00   7.972470 -3.000000e+07
2018-04-16 11:47:00   7.972470 -3.000000e+07
...                        ...           ...
2018-06-06 10:21:00  15.141100 -2.000000e+07
2018-06-06 10:22:00  15.141100 -2.000000e+07
2018-06-06 10:23:00  15.136400 -2.000000e+07
```

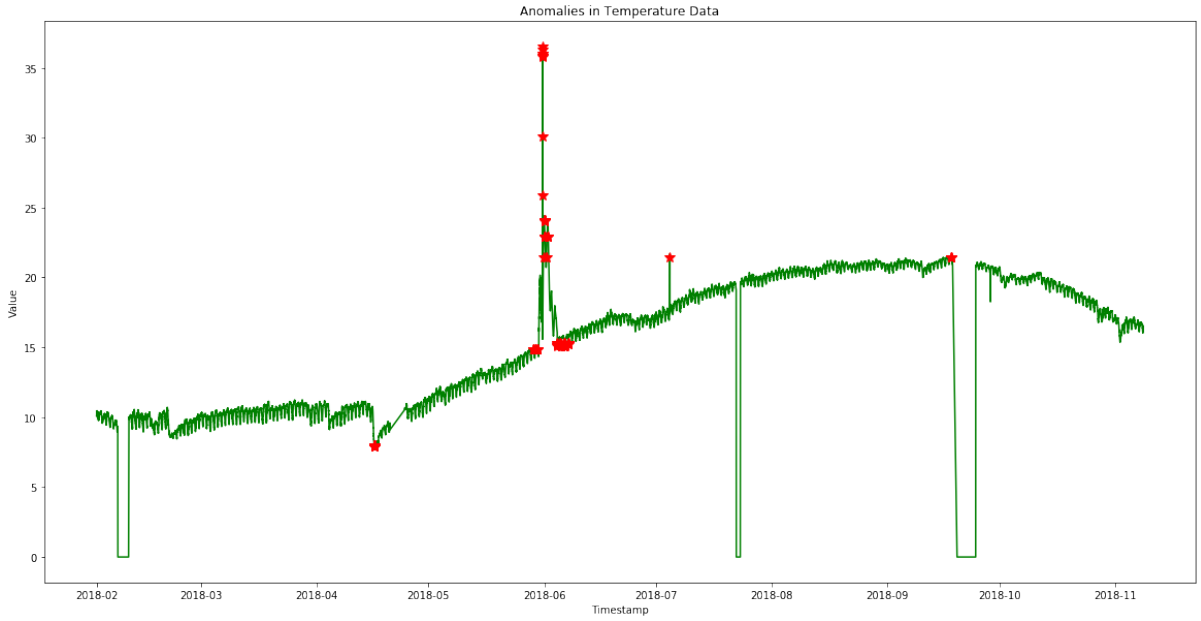
```
2018-06-06 10:28:00  15.138100 -2.000000e+07
2018-06-06 10:29:00  15.136900 -2.000000e+07
2018-06-06 10:30:00  15.136600 -2.000000e+07
2018-06-06 10:50:00  15.083500 -3.000000e+07
2018-06-06 10:51:00  15.083500 -3.000000e+07
2018-06-06 10:52:00  15.081700 -3.000000e+07
2018-06-06 10:53:00  15.079300 -3.000000e+07
2018-06-06 10:54:00  15.080500 -3.000000e+07
2018-06-06 10:55:00  15.077900 -3.000000e+07
2018-06-06 11:00:00  15.075100 -3.000000e+07
2018-06-06 12:33:00  15.078900 -3.000000e+07
2018-06-06 12:55:00  15.083300 -3.000000e+07
2018-06-06 13:11:00  15.140700 -2.000000e+07
2018-06-06 13:30:00  15.249100 -1.500000e+07
2018-06-07 10:43:00  15.277100 -1.000000e+07
2018-06-07 10:45:00  15.276700 -1.000000e+07
2018-06-07 12:25:00  15.247700 -1.500000e+07
2018-06-07 12:26:00  15.246200 -1.500000e+07
2018-06-07 12:28:00  15.254000 -1.500000e+07
2018-06-07 12:43:00  15.275300 -1.000000e+07
2018-07-04 16:28:00  21.471300 -5.000000e+07
2018-09-18 02:02:00  21.465401 -5.000000e+07
2018-09-18 02:04:00  21.470800 -5.000000e+07
2018-09-18 02:05:00  21.469999 -5.000000e+07
2018-09-18 02:06:00  21.468100 -5.000000e+07
2018-09-18 02:07:00  21.470800 -5.000000e+07
2018-09-18 02:08:00  21.466700 -5.000000e+07

[156 rows x 2 columns]
```

Export the result to csv file for further analysis.

```
In [28]: temp_anomalies.to_csv('anomalies_temp.csv')
```

Plot the result for temperature test data to deliver the whole picture.

```
In [29]: plot_lof_result(i_temp,temp_anomalies,'Anomalies in Temperature Data')
```



The red stars in the above output diagram are the instances detected as anomalies. It shows that all of these anomalies are far from other instances in the temperature test data set.

Detect anomalies on the ammonia test data as on temperature test data.

Get values of ammonia test data from dataframe:

```
In [30]: i_ammonia = process_data(ammonia)
```

```
In [31]: amm_data = reshape_training_dataset(i_ammonia)
```

Initialize an instance of LocalOutlierFactor class.

```
In [32]: clf = LocalOutlierFactor(metric='euclidean')
```

Make prediction of test data.

```
In [33]: %%time
         amm_anomalies = lof_prediction(clf,amm_data, i_ammonia)

CPU times: user 4.52 s, sys: 449 ms, total: 4.97 s
Wall time: 5.38 s
```

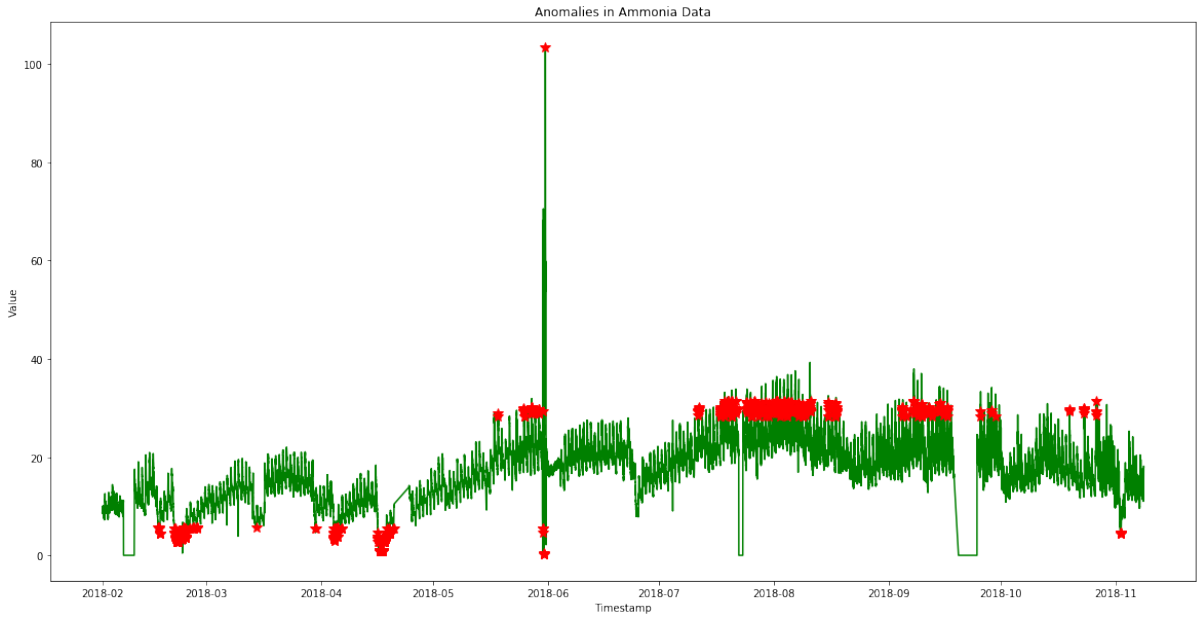
```
In [34]: print ("Number of anomalies:%s"%(amm_anomalies.shape[0]))

Number of anomalies:951
```

Export result to csv file

```
In [35]: amm_anomalies.to_csv('anomalies_amm.csv')
```

Plot the prediction result for ammonia test data.

```
In [36]: plot_lof_result(i_ammonia,amm_anomalies,'Anomalies in Ammonia Data')
```



The red stars in the above output diagram are the instances detected as anomalies. It shows that all of these anomalies are far from other instances in the ammonia test data set.

Next, detect anomalies on Chloride data set following the above steps:

```
In [37]: i_chl = process_data(chloride)
```

```
In [38]: chl_data =  reshape_traning_dataset(i_chl)
```

```
In [39]: clf = LocalOutlierFactor(n_neighbors=20, metric='euclidean',
                                  contamination=0.001)
```

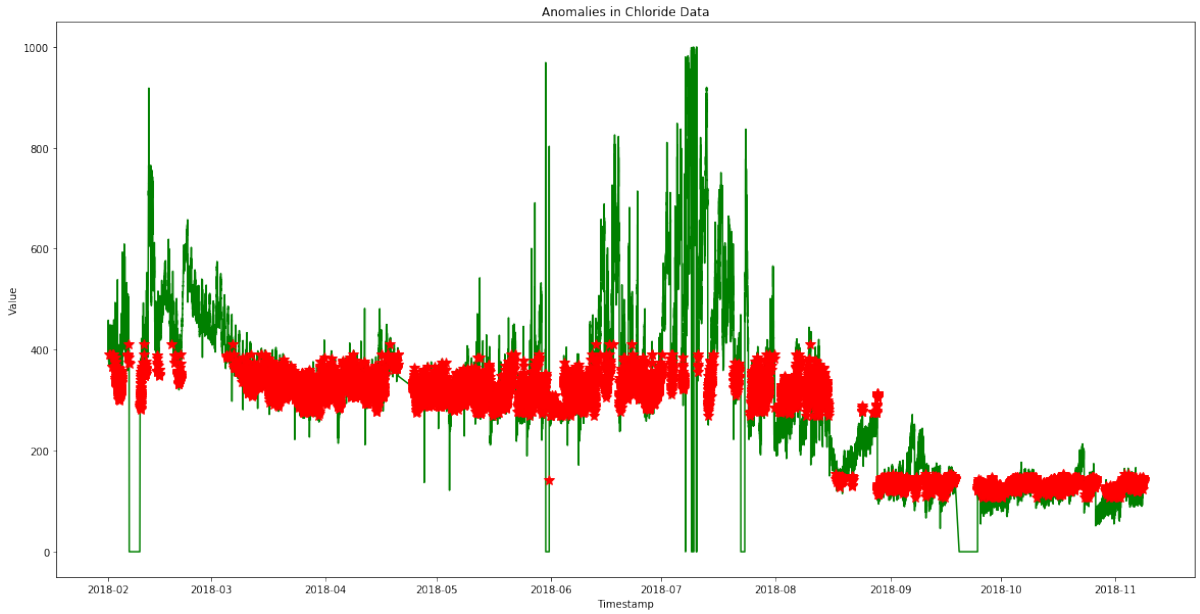
```
In [40]: %%time
         clf_anomalies = lof_prediction(clf,chl_data, i_chl)

CPU times: user 4.49 s, sys: 481 ms, total: 4.97 s
Wall time: 5.15 s
```

```
In [41]: print ("Number of Anomalies:%s"%(clf_anomalies.shape[0]))

Number of Anomalies:22407
```

Save result to csv file for further analysis.

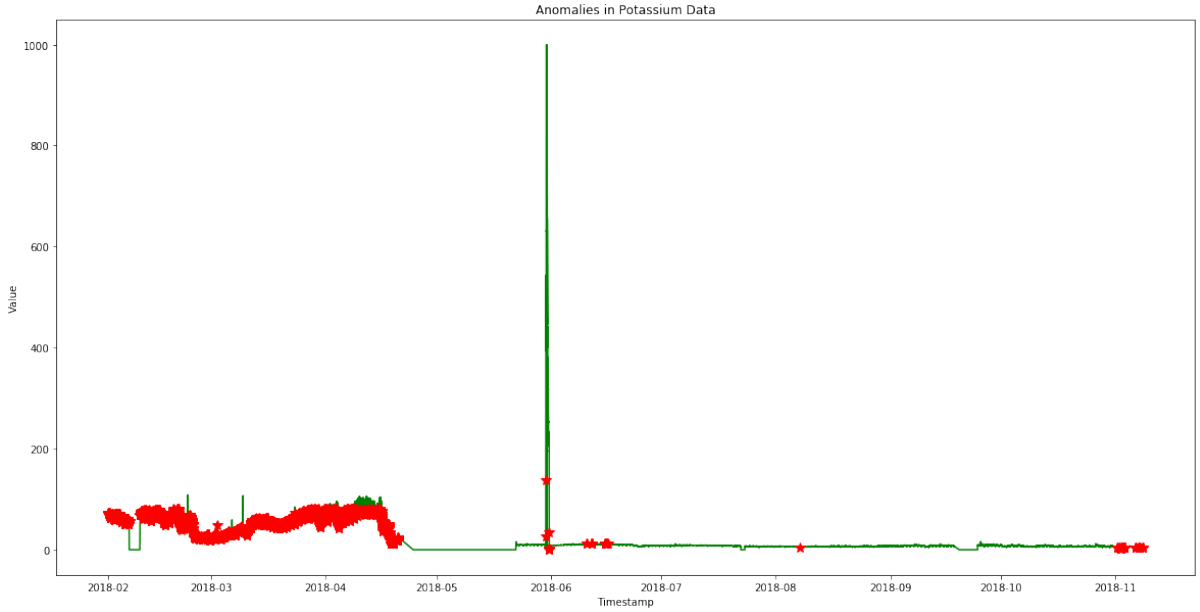
```
In [42]: clf_anomalies.to_csv('anomalies_chl.csv')
```

```
In [43]: plot_lof_result(i_chl,clf_anomalies,'Anomalies in Chloride Data')
```



Next, detect anomalies on potassium data set following the above steps:

```
In [44]: i_pota = process_data(potassium)

In [45]: pota_data = reshape_traning_dataset(i_pota)

In [46]: clf = LocalOutlierFactor(n_neighbors=20, metric='euclidean',
                                  contamination=0.001)

In [47]: %%time
         pota_anomalies = lof_prediction(clf,pota_data, i_pota)

CPU times: user 30.7 s, sys: 437 ms, total: 31.1 s
Wall time: 32.4 s

In [48]: print ("Number of Anomalies:%s"%(pota_anomalies.shape[0]))

Number of Anomalies:9316

In [49]: pota_anomalies.to_csv('anomalies_pota.csv')

In [50]: plot_lof_result(i_pota,pota_anomalies,'Anomalies in Potassium Data')
```



Statistic Approach

A Python library which rewrites Twitter's Anomaly Detection algorithms is used for testing purposes.

Try to use it to detect temperature data.

Read data from csv file.

```
In [51]: data = pd.read_csv('2018temp.csv', index_col='Timestamp',
                                parse_dates=True, squeeze=True,
         date_parser=pd_parser3)
```

Function anomaly_detect_ts is to detect anomalies in seasonal univariate time series.

```
In [52]: %%time
         results = detts.anomaly_detect_ts(data, max_anoms=0.005,
                                                direction='both', plot=False)

CPU times: user 1min 50s, sys: 311 ms, total: 1min 50s
Wall time: 2min 9s
```

Display the result.

```
In [53]: print (results)

{'anoms': Series([], dtype: float64), 'plot': None}
```

No anomalies has been found.

```
In [54]: results['anoms'].to_csv('anomalies_temp_twitter.csv')
```

Detect on ammonia data.

```
In [55]: data = pd.read_csv('2018amm.csv', index_col='Timestamp',
                                parse_dates=True, squeeze=True,
                                date_parser=pd_parser3)
```

```
In [56]: %%time
         results = detts.anomaly_detect_ts(data, max_anoms=0.005,
                                                direction='both', plot=False)
```

```
CPU times: user 1min 47s, sys: 70.4 ms, total: 1min 47s
Wall time: 1min 55s
```

Display the test result.

```
In [57]: print (results)
```

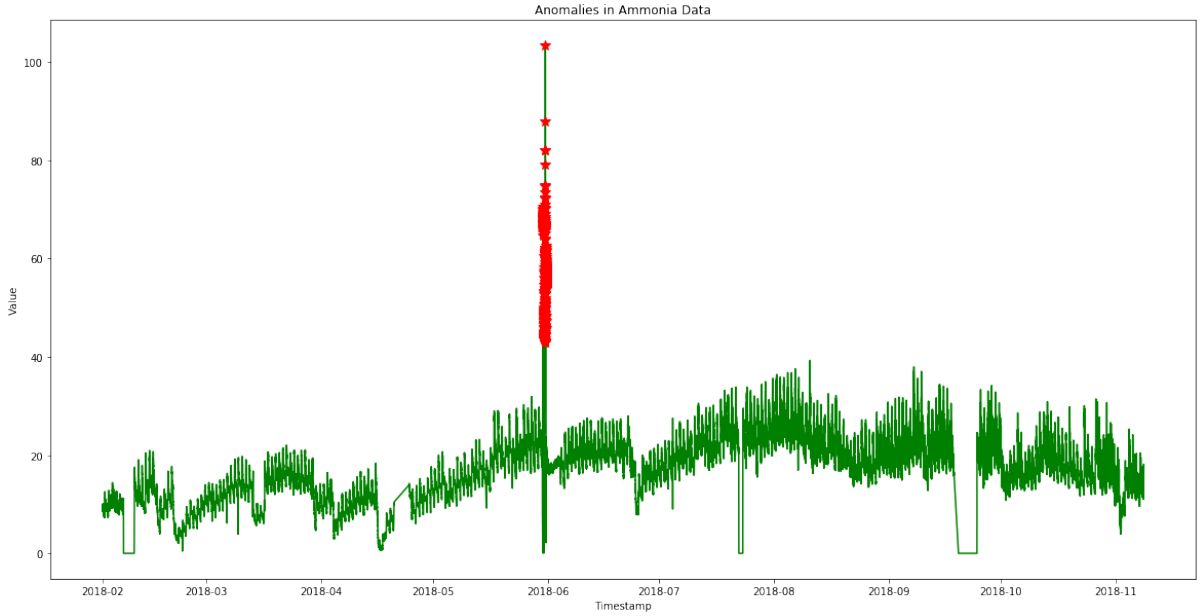
```
{'anoms': 2018-05-31 08:44:00    103.430000
2018-05-31 08:45:00     87.863701
2018-05-31 08:43:00     82.142303
2018-05-31 08:46:00     82.013199
2018-05-31 08:47:00     79.185097
2018-05-31 08:49:00     74.860603
2018-05-31 08:48:00     74.736000
2018-05-31 08:50:00     73.415001
2018-05-31 08:53:00     72.498100
2018-05-31 08:51:00     72.380302
2018-05-31 08:52:00     72.250198
2018-05-31 08:54:00     70.913803
2018-05-31 08:55:00     70.380699
2018-05-31 08:56:00     70.213600
2018-05-31 08:57:00     69.128601
2018-05-30 19:53:00     70.544800
2018-05-30 20:06:00     70.062897
2018-05-31 08:59:00     68.431503
2018-05-30 19:58:00     70.000198
2018-05-30 20:01:00     69.846802
2018-05-31 08:58:00     68.060097
2018-05-30 20:11:00     69.505501
2018-05-30 20:03:00     69.335602
2018-05-31 09:01:00     67.537102
2018-05-31 09:00:00     67.301804
2018-05-30 19:50:00     69.007004
2018-05-30 20:13:00     68.806702
2018-05-30 19:46:00     68.917900
```

```
2018-05-31 09:07:00     67.241600
2018-05-31 09:12:00     67.260101
                           ...
2018-05-30 22:02:00     44.820301
2018-05-30 21:58:00     44.739201
2018-05-30 21:53:00     44.745899
2018-05-30 21:46:00     44.616901
2018-05-30 21:57:00     44.523102
2018-05-31 07:40:00     43.741901
2018-05-30 22:03:00     44.528000
2018-05-30 22:00:00     44.411201
2018-05-30 21:49:00     44.489201
2018-05-30 22:07:00     44.329399
2018-05-30 21:50:00     44.282902
2018-05-30 21:55:00     44.254398
2018-05-30 21:52:00     44.217999
2018-05-31 07:39:00     43.372398
2018-05-30 21:48:00     44.182098
2018-05-30 21:54:00     44.143200
2018-05-30 21:44:00     44.167599
2018-05-30 22:04:00     44.054501
2018-05-30 21:47:00     44.129902
2018-05-31 04:50:00     43.393799
2018-05-30 22:11:00     43.815300
2018-05-31 04:51:00     43.254398
2018-05-31 07:38:00     43.012600
2018-05-30 21:51:00     43.792301
2018-05-30 22:06:00     43.726002
2018-05-31 04:44:00     43.231800
2018-05-31 04:49:00     43.122898
2018-05-31 07:37:00     42.979301
2018-05-30 22:13:00     43.591900
2018-05-30 21:59:00     43.655499
Length: 482, dtype: float64, 'expected': None, 'plot': None}
```

```
In [58]: results['anoms'].to_csv('anomalies_amm_twitter.csv')
```

Plot the result.

```
In [59]: plot = plt.figure(figsize=(20,10))
         plt.plot(data, color='green')
         plt.plot(results['anoms'],"r*", markersize=10)
         plt.xlabel("Timestamp")
         plt.ylabel("Value")
         plt.title("Anomalies in Ammonia Data")
         plt.show()
```



Detect on chloride data.

Read chloride data from "2018chloride.csv" file.

```
In [60]: data = pd.read_csv('2018chloride.csv', index_col='Timestamp',
                                parse_dates=True, squeeze=True,
                                date_parser=pd_parser3)

In [61]: %%time
         results = detts.anomaly_detect_ts(data, max_anoms=0.005,
                                                direction='both', plot=False)

CPU times: user 1min 34s, sys: 24.3 ms, total: 1min 34s
Wall time: 1min 39s
```

Print the detection result for chloride data.

```
In [62]: print (results)

{'anoms': 2018-07-09 18:15:00    998.767029
2018-07-09 18:04:00    998.797974
2018-07-09 18:31:00    996.416016
2018-07-09 18:34:00    996.458008
2018-07-09 18:32:00    996.500977
2018-07-09 02:27:00    998.132019
2018-07-09 02:29:00    998.101990
2018-07-09 02:36:00    998.036011
2018-07-09 18:16:00    996.278992
2018-07-09 17:59:00    998.849976
```

```
2018-07-09 18:05:00    996.174011
2018-07-09 18:03:00    996.520996
2018-07-09 02:28:00    995.614014
2018-07-09 18:02:00    996.815002
2018-07-09 18:29:00    993.869995
2018-07-09 17:55:00    996.080017
2018-07-09 17:56:00    996.067993
2018-07-09 02:21:00    993.031982
2018-07-09 02:35:00    992.926025
2018-07-09 02:34:00    992.991028
2018-07-09 02:47:00    992.487000
2018-07-09 17:54:00    993.609009
2018-07-09 02:20:00    990.486023
2018-07-09 17:57:00    993.526001
2018-07-09 18:35:00    989.020996
2018-07-10 12:38:00    999.182007
2018-07-10 12:40:00    999.466980
2018-07-09 02:33:00    990.494995
2018-07-09 12:56:00    998.926025
2018-07-09 18:25:00    988.770020
                          ...
2018-07-07 12:13:00    925.591980
2018-07-09 19:24:00    916.510010
2018-07-09 19:01:00    916.567017
2018-07-08 20:10:00    911.124023
2018-07-08 18:13:00    914.552002
2018-07-10 03:57:00    921.575012
2018-07-10 06:18:00    923.935974
2018-07-10 09:32:00    927.885986
2018-07-08 05:13:00    923.950989
2018-07-08 23:13:00    908.854980
2018-07-09 19:42:00    911.982971
2018-07-08 04:40:00    922.499023
2018-07-08 05:20:00    923.408997
2018-07-09 18:58:00    914.237976
2018-07-08 05:09:00    923.963013
2018-07-09 19:41:00    911.943970
2018-07-09 18:59:00    914.198975
2018-07-10 10:44:00    927.122986
2018-07-08 08:01:00    924.393982
2018-07-09 19:00:00    914.132019
2018-07-10 06:23:00    926.200989
2018-07-08 18:14:00    912.078003
2018-07-09 19:45:00    909.583984
2018-07-08 18:38:00    912.987000
2018-07-08 05:14:00    921.460999
```

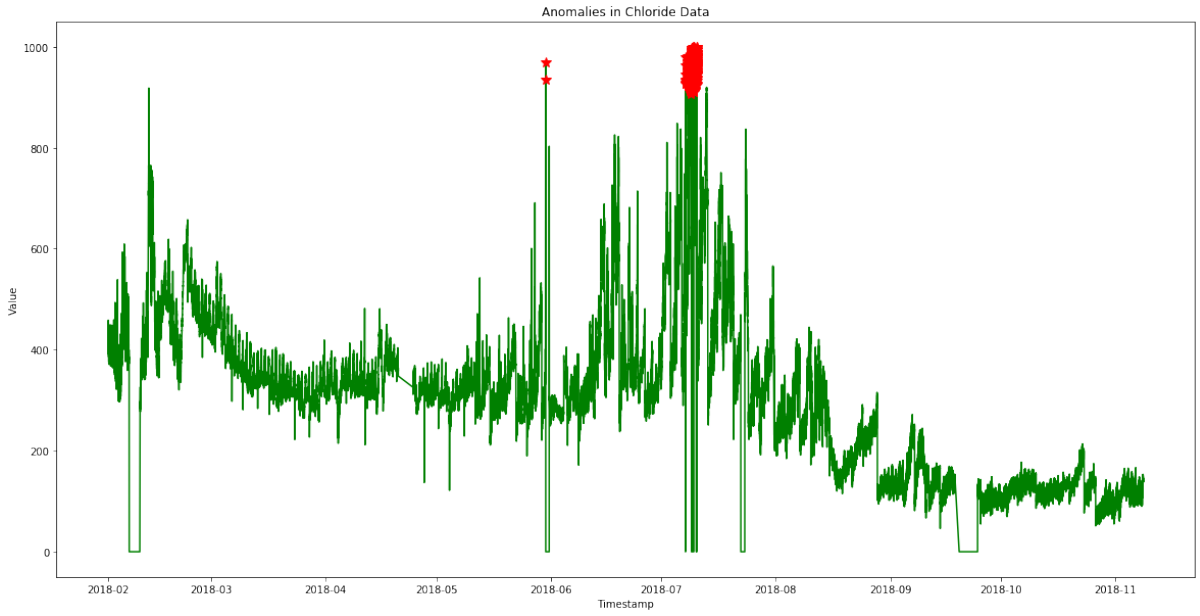
```
2018-07-09 12:25:00    921.395020
2018-07-09 19:44:00    909.546021
2018-07-10 06:20:00    921.431030
2018-07-08 23:51:00    905.762024
2018-07-10 10:43:00    924.794006
Length: 351, dtype: float64, 'expected': None, 'plot': None}
```

Save result to csv file for further analysis.

```
In [63]: results['anoms'].to_csv('anomalies_chl_twitter.csv')
```

Plot the result for chloride data.

```
In [64]: plot = plt.figure(figsize=(20,10))
         plt.plot(data, color='green')
         plt.plot(results['anoms'],"r*", markersize=10)
         plt.xlabel("Timestamp")
         plt.ylabel("Value")
         plt.title("Anomalies in Chloride Data")
         plt.show()
```



Detect on potassium data.

Read data from "2018potassium.csv" file.

```
In [65]: data = pd.read_csv('2018potassium.csv', index_col='Timestamp',
                            parse_dates=True, squeeze=True,
                            date_parser=pd_parser3)
```

```
In [66]: %%time
         results = detts.anomaly_detect_ts(data, max_anoms=0.005,
                                           direction='both', plot=False)

CPU times: user 1min 33s, sys: 60 ms, total: 1min 33s
Wall time: 1min 38s
```

Display detection result.

```
In [67]: print (results)

{'anoms': 2018-05-30 19:41:00    999.198975
2018-05-30 19:40:00    998.916992
2018-05-30 22:35:00    999.010010
2018-05-30 20:54:00    998.758972
2018-05-30 21:39:00    997.414978
2018-05-30 22:44:00    996.447998
2018-05-30 22:36:00    995.041016
2018-05-30 21:03:00    994.781006
2018-05-30 22:40:00    992.598022
2018-05-30 20:58:00    992.005981
2018-05-30 19:37:00    989.947021
2018-05-30 22:49:00    990.427979
2018-05-30 20:57:00    989.318970
2018-05-30 19:38:00    987.835022
2018-05-30 21:35:00    986.504028
2018-05-30 22:45:00    983.914978
2018-05-30 21:37:00    983.843018
2018-05-30 22:41:00    982.567017
2018-05-30 22:50:00    978.273987
2018-05-30 21:36:00    972.122009
2018-05-30 22:48:00    971.518005
2018-05-30 19:33:00    970.072021
2018-05-30 22:46:00    965.681030
2018-05-30 19:34:00    957.940002
2018-05-30 19:36:00    953.828003
2018-05-30 22:47:00    953.541016
2018-05-30 23:00:00    951.937988
2018-05-30 19:32:00    950.396973
2018-05-30 19:35:00    941.718018
2018-05-30 22:51:00    938.174011
                          ...
2018-05-31 12:08:00    204.895004
2018-05-31 12:00:00    204.848999
2018-05-31 12:52:00    204.149002
2018-05-31 12:53:00    203.886002
2018-05-31 12:56:00    203.848999
```

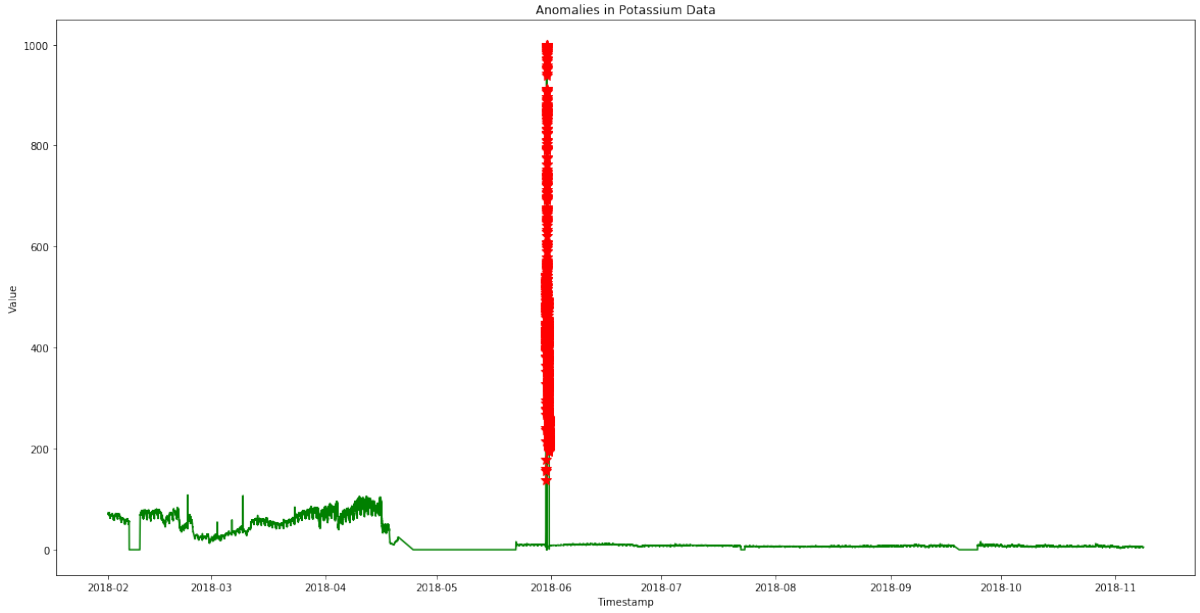
```
2018-05-31 12:54:00    203.541000
2018-05-31 12:55:00    203.115005
2018-05-31 12:57:00    202.755005
2018-05-31 12:58:00    202.673996
2018-05-31 12:59:00    202.035004
2018-05-31 13:00:00    201.106995
2018-05-31 13:01:00    199.494995
2018-05-31 13:02:00    198.410004
2018-05-31 13:12:00    197.403000
2018-05-31 13:11:00    197.078003
2018-05-31 13:10:00    196.960999
2018-05-31 13:08:00    196.438004
2018-05-31 13:14:00    196.171997
2018-05-31 13:05:00    196.160995
2018-05-31 13:09:00    196.138000
2018-05-31 13:15:00    196.026993
2018-05-31 13:04:00    195.925995
2018-05-31 13:13:00    195.602997
2018-05-31 13:07:00    195.382996
2018-05-31 13:06:00    195.248993
2018-05-31 13:03:00    195.214005
2018-05-30 16:13:00    177.501999
2018-05-30 16:12:00    158.983994
2018-05-30 16:11:00    156.643005
2018-05-30 16:10:00    137.828995
Length: 1090, dtype: float64, 'expected': None, 'plot': None}
```

Save result to csv file for further analysis.

```
In [68]: results['anoms'].to_csv('anomalies_pota_twitter.csv')
```

Plot test result for potassium data.

```
In [69]: plot = plt.figure(figsize=(20,10))
         plt.plot(data, color='green')
         plt.plot(results['anoms'],"r*", markersize=10)
         plt.xlabel("Timestamp")
         plt.ylabel("Value")
         plt.title("Anomalies in Potassium Data")
         plt.show()
```



Next, try LinkedIn Python library for time series data analysis.

First detect anomalies in temperature data.

Initialize the detector instance using test data and choose EMA algorithm.

```
In [70]: %%time
         detector = AnomalyDetector('2018temp-linkedin.csv',
                                    algorithm_name='exp_avg_detector')
```

```
CPU times: user 1min 25s, sys: 1.02 s, total: 1min 26s
Wall time: 1min 30s
```

Get the anomalies of test data set.

```
In [71]: l_temp_anomalies = detector.get_anomalies()
```

Convert timestamp to UTC date time string as timestamp in the result is epoch.

```
In [72]: l_temp_anomalies = format_luminol_result(l_temp_anomalies, False)
```

```
In [73]: l_temp_anomalies
```

```
Out[73]:    Number             Start On               End On    Exactly Happen On
         0       1  2018-07-23 15:02:00  2018-07-23 15:02:00  2018-07-23 15:02:00
         1       2  2018-09-19 17:04:00  2018-09-19 17:04:00  2018-09-19 17:04:00
         2       3  2018-09-24 17:31:00  2018-09-24 17:31:00  2018-09-24 17:31:00
```

The result of Luminol actually gives the start time of anomaly and end time of anomaly as well.

```
In [74]: temp_anomalies.to_csv('anomalies_temp_linkedin.csv')
```

Test Luminol on ammonia data following the above steps.

```
In [75]: %%time
         detector = AnomalyDetector('2018amm-linkedin.csv',
                                    algorithm_name='exp_avg_detector')

CPU times: user 1min 27s, sys: 1.13 s, total: 1min 28s
Wall time: 1min 32s
```

```
In [76]: l_amm_anomalies = detector.get_anomalies()
```

Timestamp in the result is epoch, now convert it to UTC date time string.

```
In [77]: l_amm_anomalies = format_luminol_result(l_amm_anomalies, False)

In [78]: l_amm_anomalies
```

Out[78]:

	Number	Start On	End On	Exactly Happen On
0	1	2018-05-30 20:39:00	2018-05-30 20:39:00	2018-05-30 20:39:00
1	2	2018-05-30 20:42:00	2018-05-30 20:44:00	2018-05-30 20:44:00
2	3	2018-05-30 21:44:00	2018-05-30 21:44:00	2018-05-30 21:44:00
3	4	2018-05-30 21:54:00	2018-05-30 21:54:00	2018-05-30 21:54:00
4	5	2018-05-30 21:57:00	2018-05-30 22:00:00	2018-05-30 22:00:00
5	6	2018-05-30 22:03:00	2018-05-30 22:03:00	2018-05-30 22:03:00
6	7	2018-05-30 22:35:00	2018-05-30 22:37:00	2018-05-30 22:37:00
7	8	2018-05-30 22:39:00	2018-05-30 22:39:00	2018-05-30 22:39:00
8	9	2018-05-30 23:36:00	2018-05-30 23:36:00	2018-05-30 23:36:00
9	10	2018-05-30 23:40:00	2018-05-30 23:41:00	2018-05-30 23:41:00
10	11	2018-05-30 23:44:00	2018-05-30 23:44:00	2018-05-30 23:44:00
11	12	2018-05-31 01:43:00	2018-05-31 01:43:00	2018-05-31 01:43:00
12	13	2018-05-31 09:43:00	2018-05-31 09:44:00	2018-05-31 09:44:00
13	14	2018-05-31 14:16:00	2018-05-31 14:17:00	2018-05-31 14:17:00
14	15	2018-05-31 14:36:00	2018-05-31 14:38:00	2018-05-31 14:37:00
15	16	2018-07-23 15:02:00	2018-07-23 15:02:00	2018-07-23 15:02:00

```
In [79]: l_amm_anomalies.to_csv('anomalies_amm_linkedin.csv')
```

Test Luminol on chloride data following the above steps.

```
In [80]: %%time
         detector = AnomalyDetector('2018chloride-linkedin.csv',
                                    algorithm_name='exp_avg_detector')

CPU times: user 1min 27s, sys: 1.1 s, total: 1min 28s
Wall time: 1min 32s
```

```
In [81]: l_chl_anomalies = detector.get_anomalies()
```

Timestamp in the result is epoch, now convert it to UTC date time string.

```
In [82]: l_chl_anomalies = format_luminol_result(l_chl_anomalies, False)
```

Display the test result.

```
In [83]: l_chl_anomalies
```

Out[83]:

	Number	Start On	End On	Exactly Happen On
0	1	2018-05-30 15:13:00	2018-05-30 15:14:00	2018-05-30 15:14:00
1	2	2018-05-31 14:21:00	2018-05-31 14:22:00	2018-05-31 14:22:00
2	3	2018-07-07 13:17:00	2018-07-07 13:18:00	2018-07-07 13:18:00
3	4	2018-07-09 03:24:00	2018-07-09 03:26:00	2018-07-09 03:26:00
4	5	2018-07-09 03:37:00	2018-07-09 03:39:00	2018-07-09 03:39:00
5	6	2018-07-09 03:47:00	2018-07-09 03:48:00	2018-07-09 03:48:00
6	7	2018-07-09 03:55:00	2018-07-09 03:56:00	2018-07-09 03:56:00
7	8	2018-07-09 07:22:00	2018-07-09 07:24:00	2018-07-09 07:24:00
8	9	2018-07-09 07:29:00	2018-07-09 07:29:00	2018-07-09 07:29:00
9	10	2018-07-09 13:57:00	2018-07-09 13:59:00	2018-07-09 13:59:00
10	11	2018-07-09 18:54:00	2018-07-09 18:56:00	2018-07-09 18:56:00
11	12	2018-07-09 18:58:00	2018-07-09 18:58:00	2018-07-09 18:58:00
12	13	2018-07-09 19:00:00	2018-07-09 19:00:00	2018-07-09 19:00:00
13	14	2018-07-09 19:02:00	2018-07-09 19:02:00	2018-07-09 19:02:00
14	15	2018-07-09 19:06:00	2018-07-09 19:07:00	2018-07-09 19:07:00
15	16	2018-07-09 19:15:00	2018-07-09 19:16:00	2018-07-09 19:16:00
16	17	2018-07-09 19:27:00	2018-07-09 19:28:00	2018-07-09 19:28:00
17	18	2018-07-10 12:10:00	2018-07-10 12:12:00	2018-07-10 12:12:00
18	19	2018-07-10 12:57:00	2018-07-10 12:59:00	2018-07-10 12:59:00
19	20	2018-07-10 13:38:00	2018-07-10 13:38:00	2018-07-10 13:38:00
20	21	2018-07-10 13:40:00	2018-07-10 13:40:00	2018-07-10 13:40:00
21	22	2018-07-10 13:43:00	2018-07-10 13:44:00	2018-07-10 13:44:00
22	23	2018-07-10 13:46:00	2018-07-10 13:46:00	2018-07-10 13:46:00
23	24	2018-07-10 13:59:00	2018-07-10 13:59:00	2018-07-10 13:59:00

Save result to csv file.

```
In [84]: l_chl_anomalies.to_csv('anomalies_chloride_linkedin.csv')
```

Test Luminol on potassium data following the same steps.

```
In [85]: %%time
         detector = AnomalyDetector('2018potassium-linkedin.csv',
                                    algorithm_name='exp_avg_detector')

CPU times: user 1min 25s, sys: 973 ms, total: 1min 26s
Wall time: 1min 31s
```

```
In [86]: l_pot_anomalies = detector.get_anomalies()
```

Timestamp in the result is epoch, now convert it to UTC datatime string.

```
In [87]: l_pot_anomalies = format_luminol_result(l_pot_anomalies, False)
```

Display the test result.

```
In [88]: l_pot_anomalies
```

Out[88]:

	Number	Start On	End On	Exactly Happen On
0	1	2018-05-30 20:39:00	2018-05-30 20:50:00	2018-05-30 20:40:00
1	2	2018-05-30 21:54:00	2018-05-30 22:08:00	2018-05-30 21:55:00
2	3	2018-05-30 22:35:00	2018-05-30 22:46:00	2018-05-30 22:38:00
3	4	2018-05-30 23:35:00	2018-05-30 23:50:00	2018-05-30 23:37:00
4	5	2018-05-31 14:16:00	2018-05-31 14:17:00	2018-05-31 14:17:00

Save result to csv file for further analysis.

```
In [89]: l_pot_anomalies.to_csv('anomalies_pota_linkedin.csv')
```

Chapter 4

Analysis and Discussion

The test results have been extracted from the previous chapter. The data set for the primary experiment was manually analyzed to identify all the anomalies and then each anomaly is checked to decide if they are contextual anomalies.

4.1 Evaluation I

4.1.1 Temperature Data

Techniques	Anomalies	False Positives	False Negatives
Local Outlier Factor	26	0	0
S-H-ESD	1	0	25
EMA	10 time ranges	2	12

Table 4.1: Experiment Result for Temperature

Note: For EMA, 8 time ranges cover 14 outliers.

Machine Learning

Local Outlier Factor has been used in the experiment to detect anomaly for temperature data.

Figure 4.1 shows the position of all the anomaly instances in the test data set. The same 26 anomaly instances listed in Table 4.2 were found under all the test scenarios. The results are consistent and reproducible.

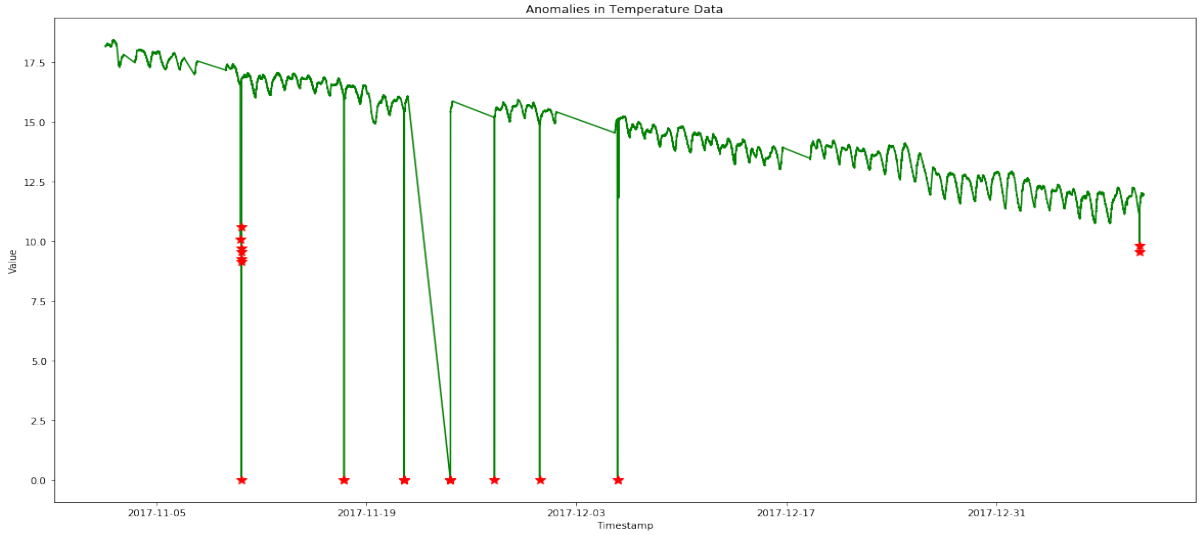


FIGURE 4.1: Chart of LOF Result for Temperature

Timestamp	Value	LOF
2017-11-10 14:38:00	10.0747	114.0423
2017-11-10 15:43:00	10.6043	31.1438
2017-11-10 15:44:00	9.2604	205.5421
2017-11-10 15:45:00	0.0000	440.1642
2017-11-10 15:46:00	9.7064	152.8599
2017-11-10 15:47:00	9.1378	220.3304
2017-11-10 15:48:00	9.5711	168.7108
2017-11-17 12:08:00	0.0000	440.1642
2017-11-17 12:09:00	0.0000	440.1642
2017-11-21 12:26:00	0.0000	440.1642
2017-11-21 12:27:00	0.0000	440.1642
2017-11-21 12:28:00	0.0000	440.1642
2017-11-21 12:29:00	0.0000	440.1642
2017-11-21 12:30:00	0.0000	440.1642
2017-11-24 14:19:00	0.0000	440.1642
2017-11-24 14:20:00	0.0000	440.1642
2017-11-24 14:21:00	0.0000	440.1642
2017-11-24 14:22:00	0.0000	440.1642
2017-11-24 14:23:00	0.0000	440.1642
2017-11-24 14:24:00	0.0000	440.1642
2017-11-27 12:52:00	0.0000	440.1642
2017-11-30 13:13:00	0.0000	440.1642
2017-12-05 17:39:00	0.0000	440.1642
2017-12-05 17:47:00	0.0000	440.1642
2018-01-09 13:19:00	9.8053	141.2707
2018-01-09 13:20:00	9.5497	171.2252

TABLE 4.2: LOF Result for Temperature

Now "zoom in" to the time period when an individual anomaly occurs using Microsoft Excel to verify each outlier. The red dots in the figures denote the anomalies detected by LOF.

Figure 4.2 shows the exact position of the first anomaly instance. The figure indicates that this single instance with value equals 10.0747 is far from its closest neighbors whose values are approximately 16.5. Thus, it is an anomaly.

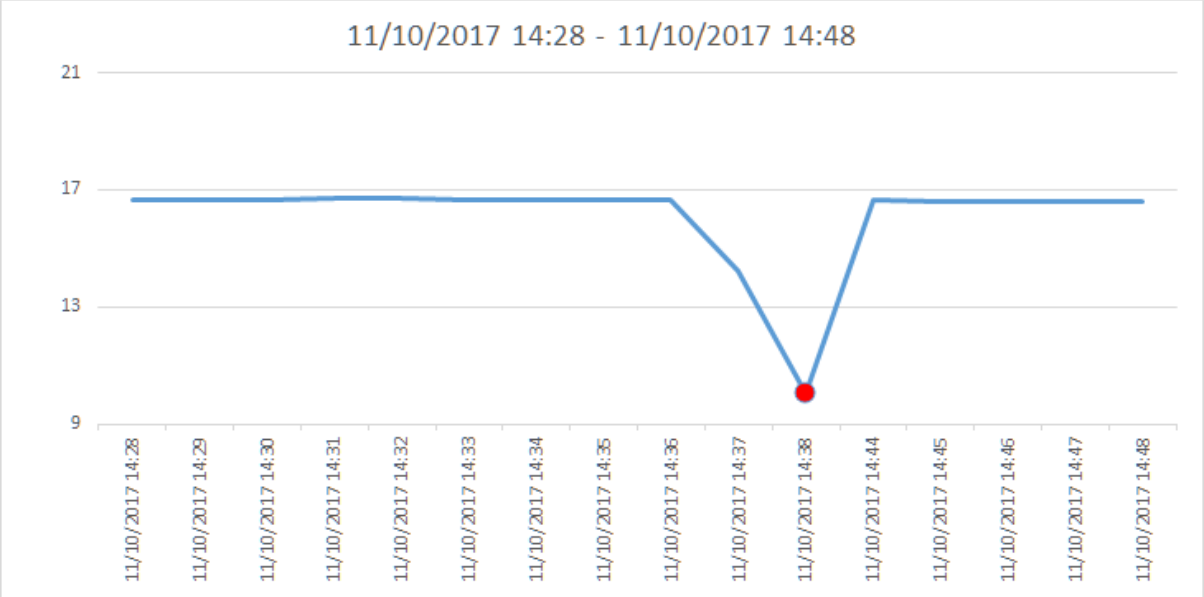


FIGURE 4.2: Temperature on Nov 10

Figure 4.3 shows the exact positions of the second to the seventh anomaly instances. The figure indicates that these six instances with value less than 11 are far from their closest neighbors whose values are approximately 16.5. Thus, they are anomalies.

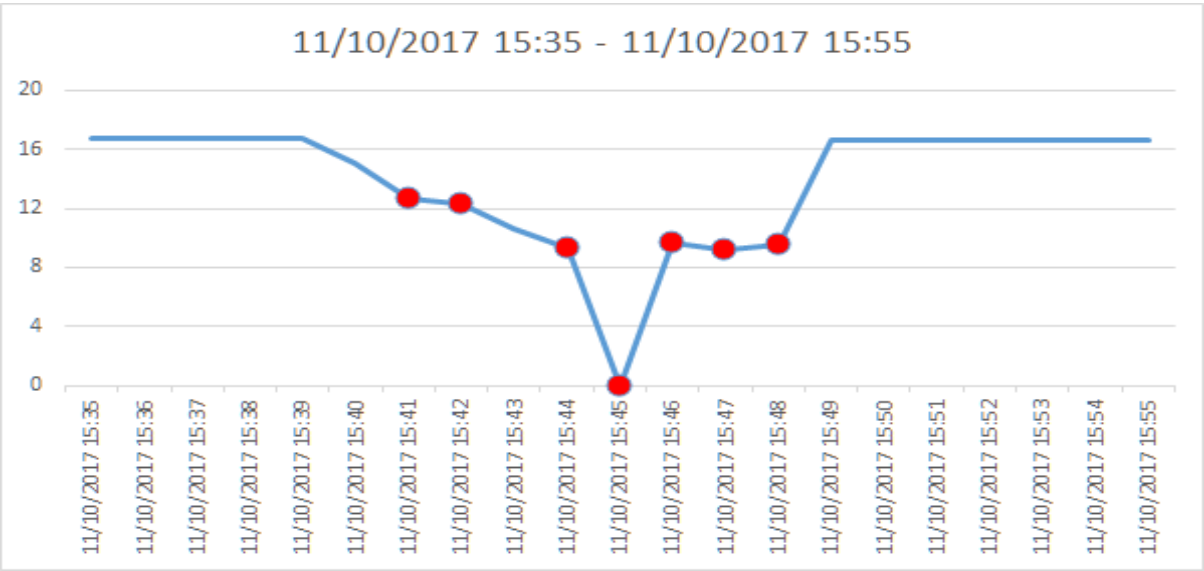


FIGURE 4.3: Temperature on Nov 10 afternoon

Figure 4.4 shows the exact positions of the eighth to the ninth anomaly instances. The figure indicates that these two instances with values of zero are far from their closest neighbors whose values are approximately 16. Thus, they are anomalies.

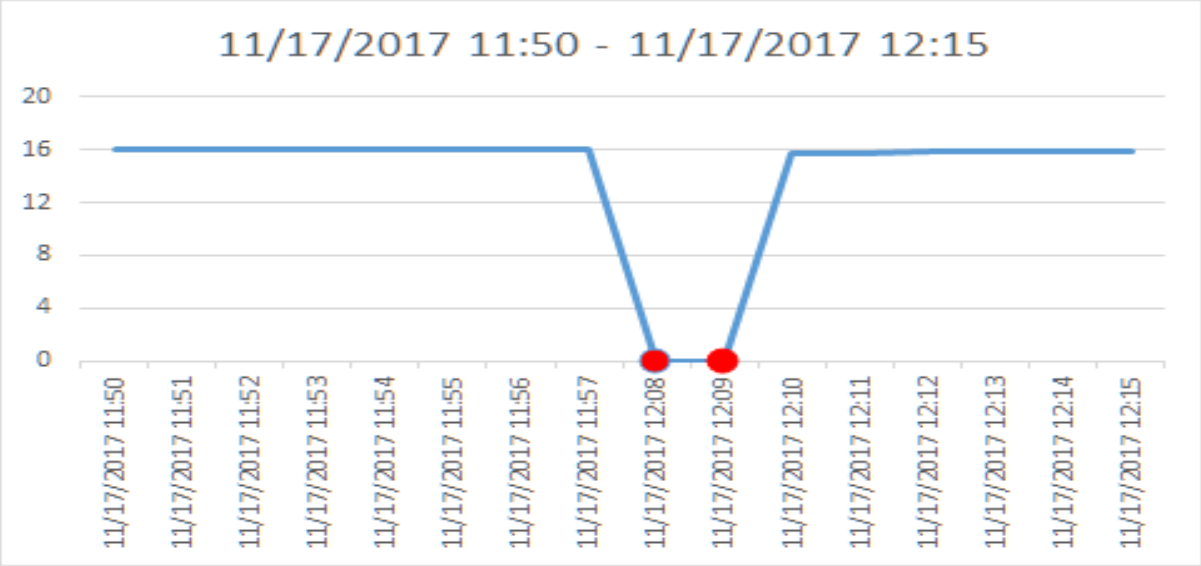


FIGURE 4.4: Temperature on Nov 17

Figure 4.5 shows the exact positions of the tenth to the fourteenth anomaly instances. The figure indicates that these five instances with values of zero are far from their closest neighbors whose values are approximately 15.5. Thus, they are anomalies.

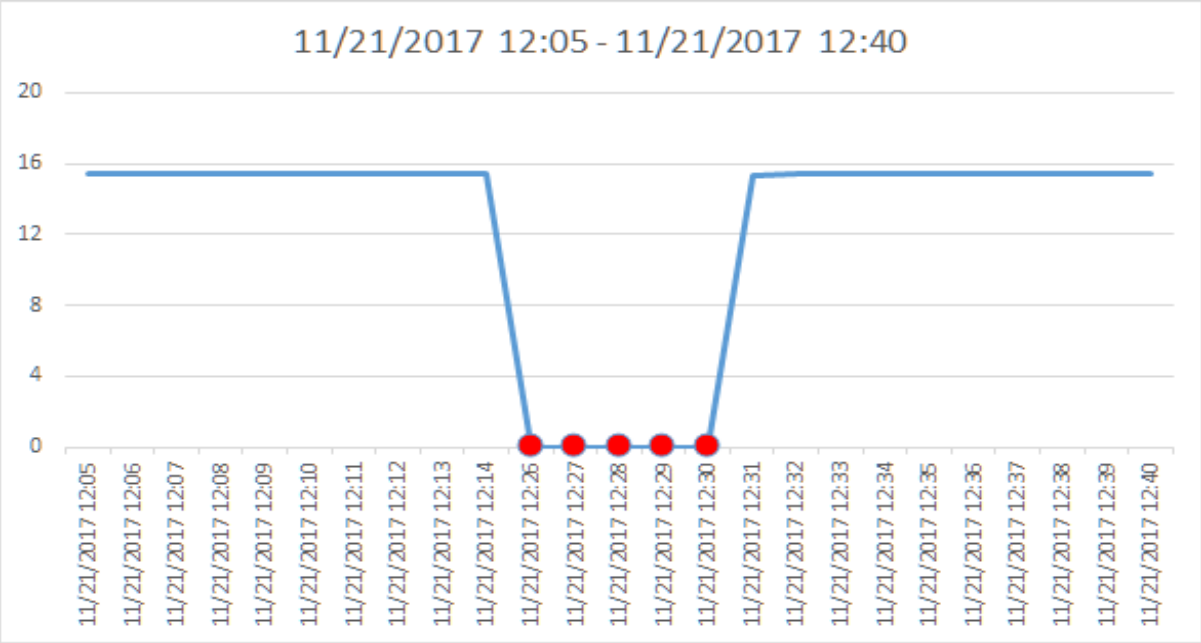


FIGURE 4.5: Temperature on Nov 21

Figure 4.6 shows the exact positions of the fifteenth to the twentieth anomaly instances. The figure indicates that these six instances with values of zero are far from their closest neighbors whose values are approximately 15.5. Thus, they are anomalies.

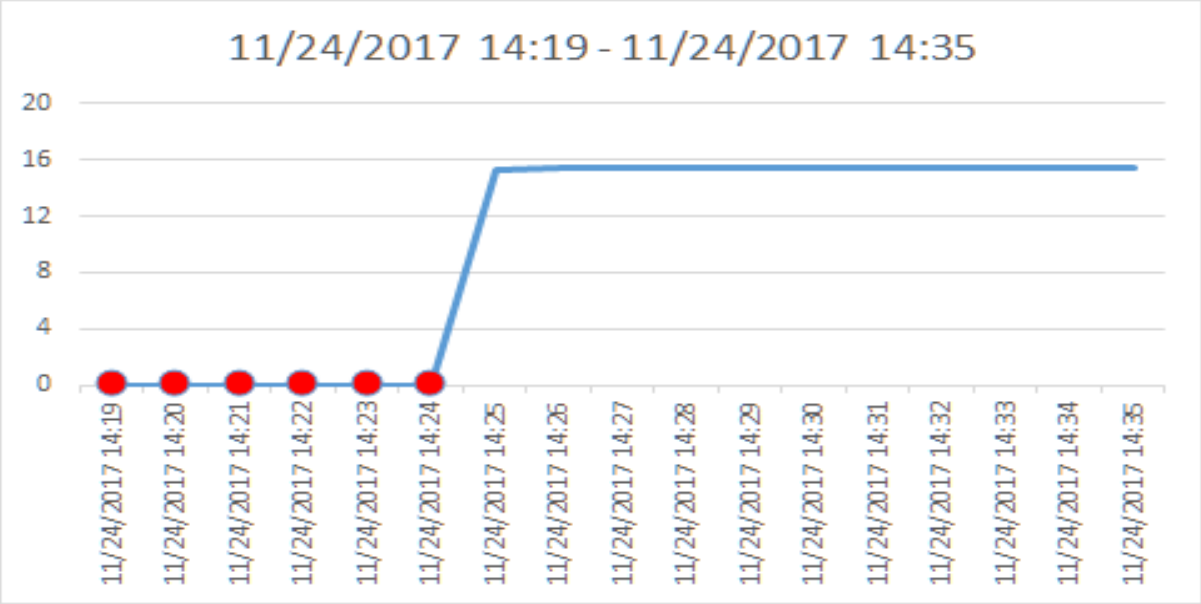


FIGURE 4.6: Temperature on Nov 24

Figure 4.7 shows the exact position of the twenty-first anomaly instance. The figure indicates that this single instance with value of zero is far from its closest neighbors whose values are approximately 15. Thus, it is an anomaly.

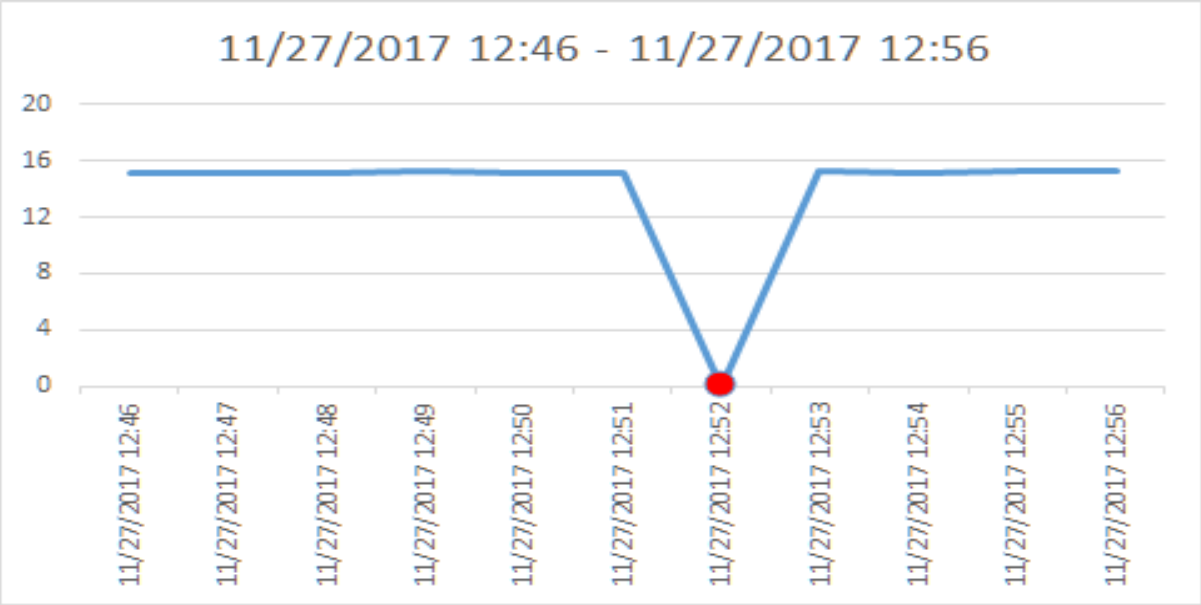


FIGURE 4.7: Temperature on Nov 27

Figure 4.8 shows the exact position of the twenty-second anomaly instance. The figure indicates that this single instance with value of zero is far from its closest neighbors whose values are approximately 15. Thus, it is an anomaly.

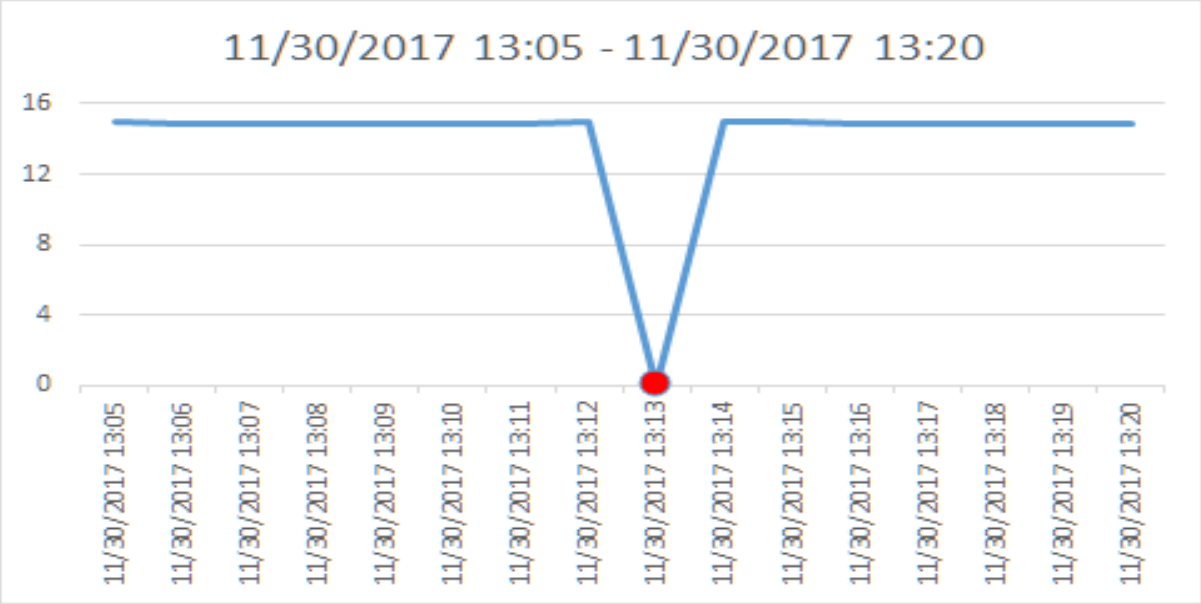


FIGURE 4.8: Temperature on Nov 30

Figure 4.9 shows the exact positions of the twenty-third to the twenty-fourth anomaly instances. The figure indicates that these two instances with values of zero are far from their closest neighbors whose values are approximately 15. Thus, they are anomalies.

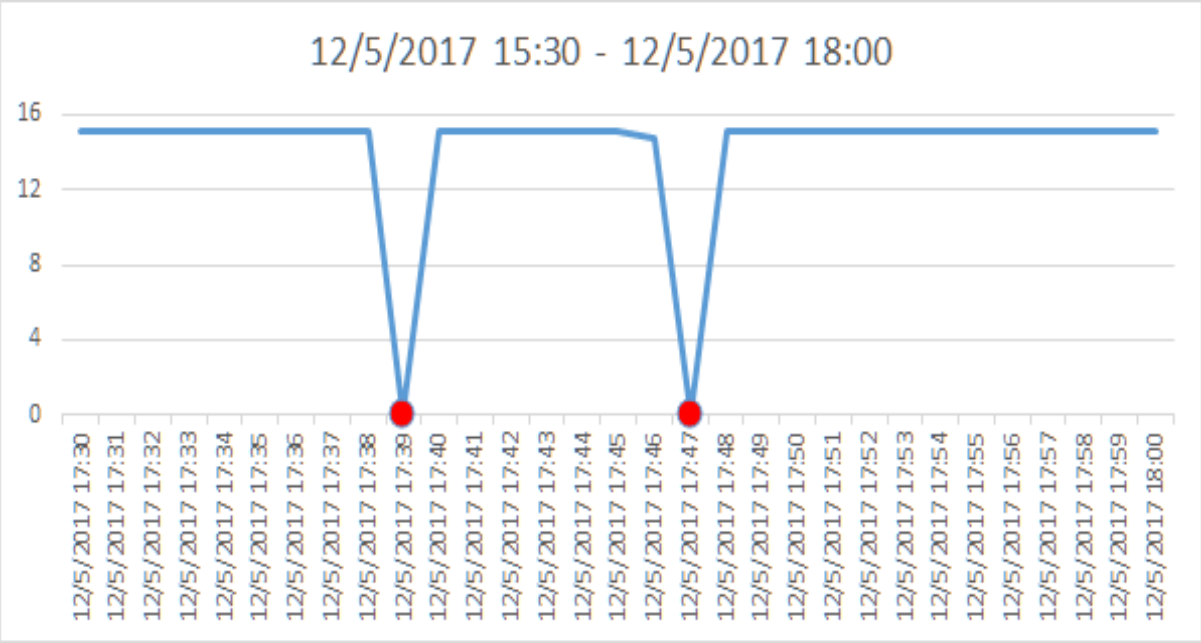


FIGURE 4.9: Temperature on Dec 05

Figure 4.10 shows the exact positions of the twenty-fifth to the twenty-sixth anomaly instances. The figure indicates that these two instances with values of 9.8053 and 9.5497 respectively are far from their closest neighbors whose values are approximately 15. Thus, they are anomalies.

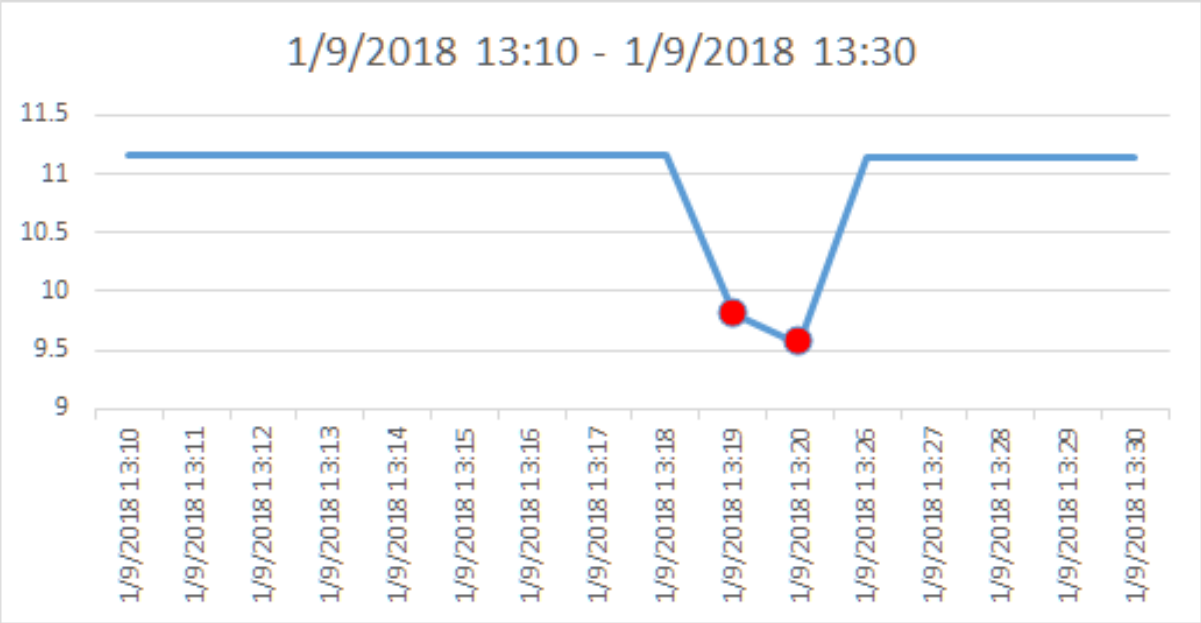


FIGURE 4.10: Temperature on Jan 09

Isolation Forest (Section 2.3.7) is applied to detect anomalies in the temperature data in Experiment 3.3.2. Over 7000 points were marked as anomalies using 0.6 suggested by the author, and it is very difficult to find a feasible threshold for improvement. For instance, most of the anomalies are considered as normal data if 0.7 is set as threshold, but hundreds of normal points are considered as anomalies if 0.65 is set as threshold.

Robust Random Cut Forest (Section 2.3.8) is used to detect anomalies on temperature in Experiment 3.3.2. It took almost 12 minutes for the detection which is longer than the other algorithms like LOF which only took hundreds of milliseconds. Some outliers are marked with low anomaly scores (like the zeros) whereas some normal points are marked with high anomaly scores which makes it difficult to find a good threshold to filter out anomalies.

Statistic Approach

S-H-ESD only gets one anomaly point as show on Table 4.3. Figure 4.6 shows the detail of the relationship between this single anomaly with its close neighbors. One of the potentially reasons why S-H-ESD could not find other outliers is because those outliers seem periodic in the test data set and thus S-H-ESD considers those outliers are normal data.

Timestamp	Value
2017-11-24 14:20:00	0

TABLE 4.3: S-H-ESD Result for Temperature

As shown in Figure 4.4, even though EMA found ten time ranges for anomalies, eight of them covered 14 abnormal points. The data in the other two ranges are normal data. This is because they are after consecutive anomaly points, and EMA considers them as abnormal based on the way EMA is calculated (Section 2.4.3).

Anomaly Number	Start On	End On	Exactly Happen on
1	2017-11-10 15:45:00	2017-11-10 15:45:00	2017-11-10 15:45:00
2	2017-11-17 12:08:00	2017-11-17 12:09:00	2017-11-17 12:09:00
3	2017-11-21 12:26:00	2017-11-21 12:29:00	2017-11-21 12:29:00
4	2017-11-21 12:31:00	2017-11-21 12:32:00	2017-11-21 12:32:00
5	2017-11-24 14:19:00	2017-11-24 14:22:00	2017-11-24 14:22:00
6	2017-11-24 14:25:00	2017-11-24 14:26:00	2017-11-24 14:26:00
7	2017-11-27 12:52:00	2017-11-27 12:52:00	2017-11-27 12:52:00
8	2017-11-30 13:13:00	2017-11-30 13:13:00	2017-11-30 13:13:00
9	2017-12-05 17:39:00	2017-12-05 17:39:00	2017-12-05 17:39:00
10	2017-12-05 17:47:00	2017-12-05 17:47:00	2017-12-05 17:47:00

TABLE 4.4: EMA Result for Temperature

4.1.2 Ammonia Data

All three approaches successfully detected the same four anomalies in the ammonia data. Figure 4.11 and Figure 4.12 illustrate the exact positions for those four anomaly instances.

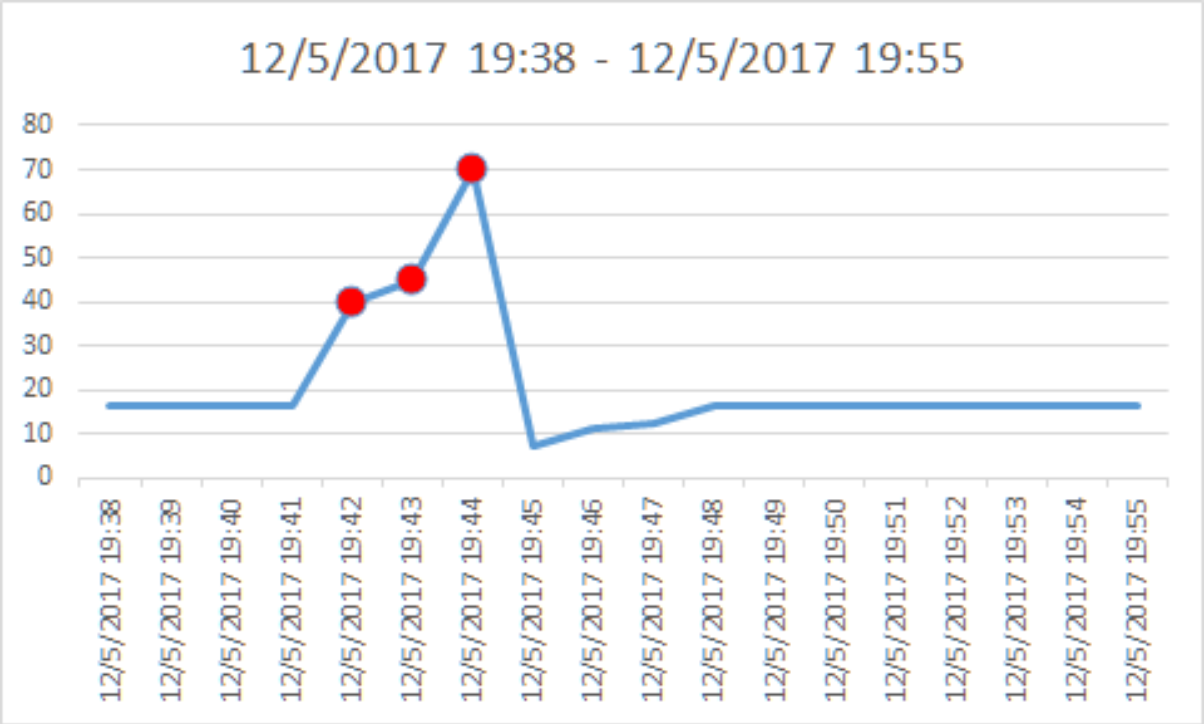


FIGURE 4.11: Ammonia on Dec 05

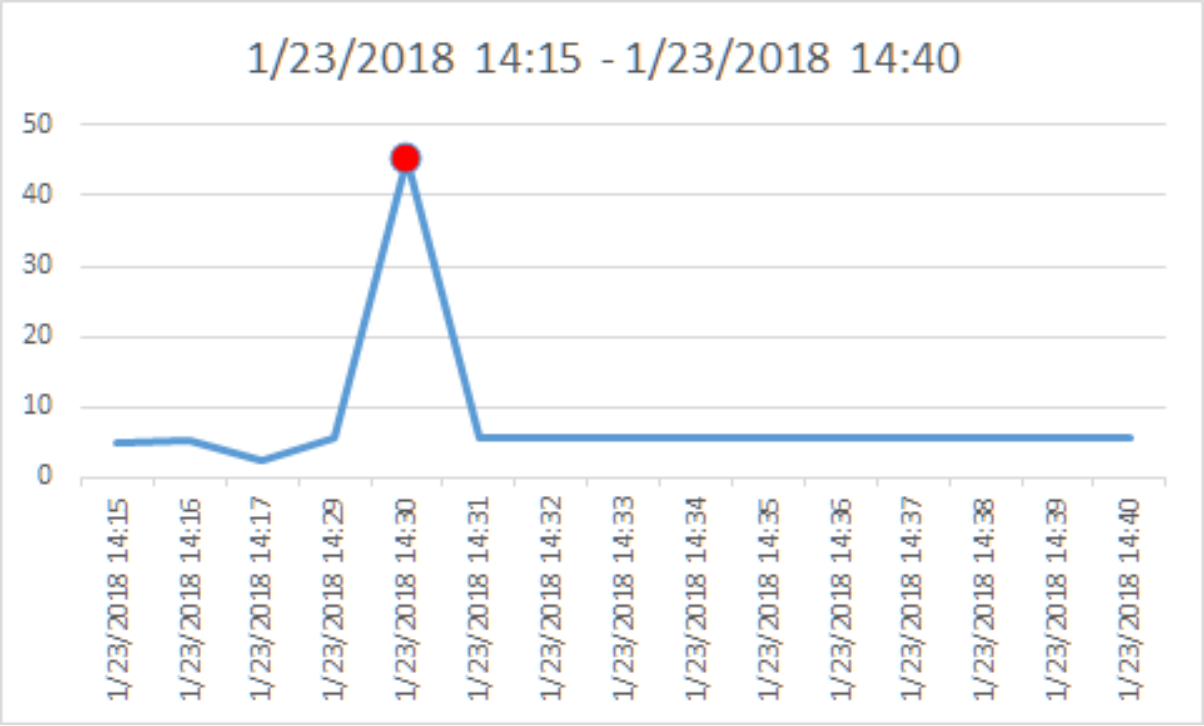


FIGURE 4.12: Ammonia on Jan 23

Machine Learning

In the experiment using Local Outlier Factor, the same four anomaly instances listed in Table 4.5 were found using all the test scenarios. The result is consistent and reproducible.

Figure 4.13 illustrates the positions of all the anomaly instances in the test data set.

Timestamp	Value	Is Inlier
2017-12-05 19:42:00	39.7006	-61.4179
2017-12-05 19:43:00	44.7568	-81.6157
2017-12-05 19:44:00	69.9964	-177.8467
2018-01-23 14:30:00	45.3118	-83.8327

TABLE 4.5: LOF Result for Ammonia

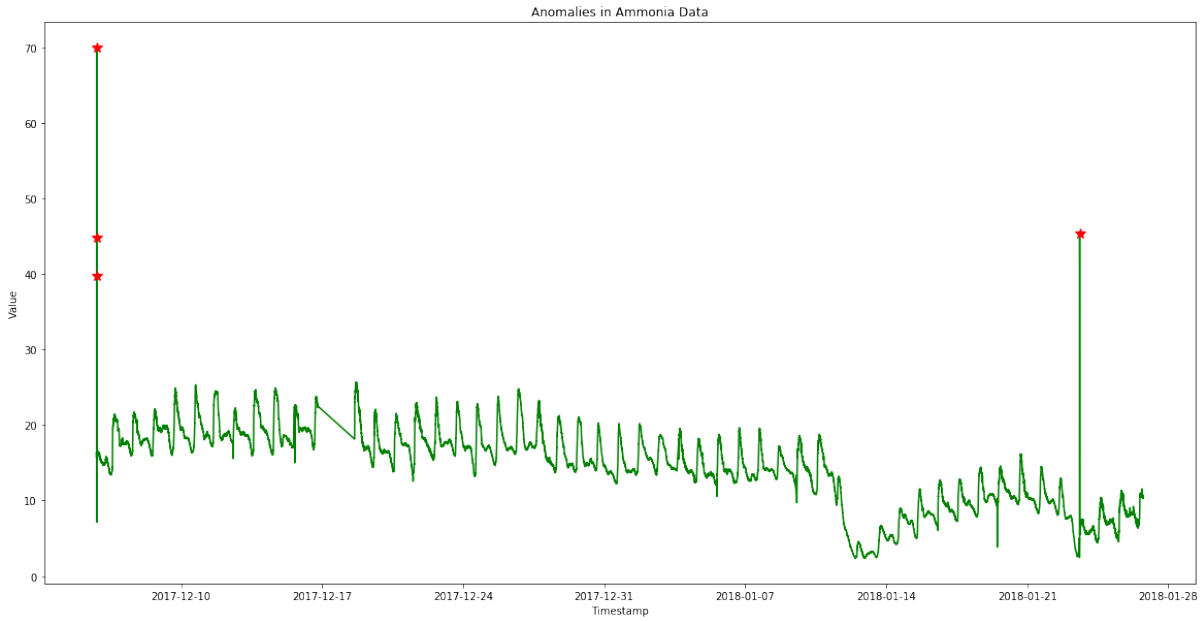


FIGURE 4.13: Chart of LOF Result for Ammonia

Isolation Forest (Section 2.3.7) is applied to detect anomalies in the ammonia data in Experiment 3.3.2. Over 7000 points for ammonia data marked as anomalies using 0.6 suggested by the author, and it is very difficult to find a feasible threshold for improvement. For instance, most of the anomalies are considered as normal data if 0.7 is set as threshold, but hundreds of normal points are still considered as anomalies if 0.65 is set as threshold.

Robust Random Cut Forest (Section 2.3.8) is used to detect anomalies on ammonia in Experiment 3.3.2. It took over 15 minutes for detection whereas LOF only took hundreds of milliseconds. It is difficult to find a good threshold to filter out anomalies as some outliers are marked with low anomaly scores whereas some normal points are marked with high anomaly scores.

Statistic Approach

Figure 4.6 shows the four anomalies detected by S-H-ESD which are the same with as those detected using LOF detection.

Timestamp	Value
2017-12-05 19:44:00	69.9964
2018-01-23 14:30:00	45.3118
2017-12-05 19:43:00	44.7568
2017-12-05 19:42:00	39.7006

TABLE 4.6: S-H-ESD Result for Ammonia

As shown on Figure 4.7, EMA could not clearly identify each abnormal point if they are consecutive points. The results only give a range for when the anomaly happened. However, those ranges cover the same anomalies detected by LOF and S-H-ESD.

Anomaly Number	Start On	End On	Exactly Happen on
1	2017-12-05 19:42:00	2017-12-05 19:45:00	2017-12-05 19:44:00
2	2018-01-23 14:30:00	2018-01-23 14:30:00	2018-01-23 14:30:00

TABLE 4.7: EMA Result for Ammonia

4.1.3 Execution Time

Table 4.8 shows the execution time to detect anomalies in the water temperature data set. For a data set with 78,338 records, LOF took 890 milliseconds to detect anomalies. S-H-ESD needed 3.79 seconds for detection. And, EMA took 19.4 seconds.

Number of Records	LOF	S-H-ESD	EMA
78,338	890 ms	3.79 s	20s

TABLE 4.8: Execution Time for Temperature

Table 4.9 shows the execution time to detect anomalies in the ammonia data set. For a data set with 71,926 records, LOF took 785 milliseconds to detect anomalies. S-H-ESD needed 3.25 seconds for detection and EMA took 16.7 seconds.

Sensors	LOF	S-H-ESD	EMA
Ammonia	785 ms	3.25 s	17.4 s

TABLE 4.9: Execution Time for Ammonia

The result of the experiment confirms that LOF is much faster and more efficient than S-H-ESD and EMA when they are used to detect anomalies in this water temperature and ammonia data.

4.2 Evaluation II

There are over 70,000 records in each data set in previous experiment section. The machine learning and statistic learning approaches have been evaluated with large volume data following the same steps as well.

In the evaluation, another four large data sets were used, each having 395,715 records.

4.2.1 Results

Table 4.10 shows the result of the experiment on the four large data sets.

Parameters	LOF	S-H-ESD	EMA
Temperature	156	0	3
Ammonia	951	422	28
Chloride	22407	359	47
Potassium	9317	1090	27

TABLE 4.10: Number of Anomalies Detected by Each Technique

Figure 4.14 illustrates the positions of all the anomaly instances detected by LOF in the test temperature data set.

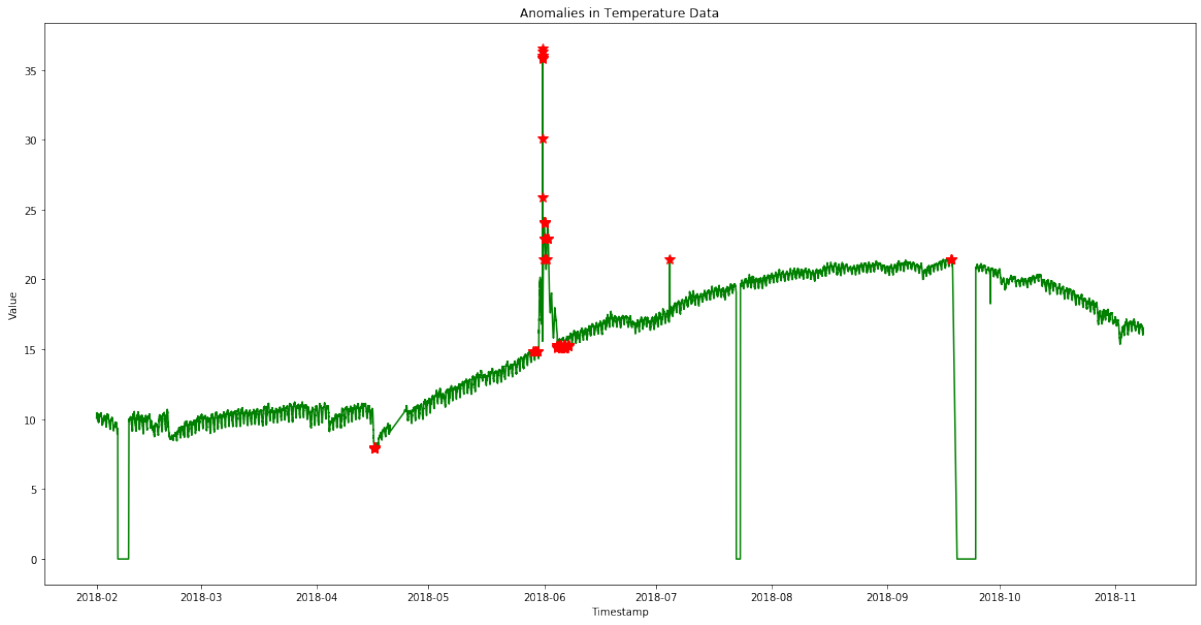


FIGURE 4.14: Chart of LOF Result for Water Temperature 2018

Figure 4.15 illustrates the positions of all the anomaly instances detected by LOF in the ammonia test data set.

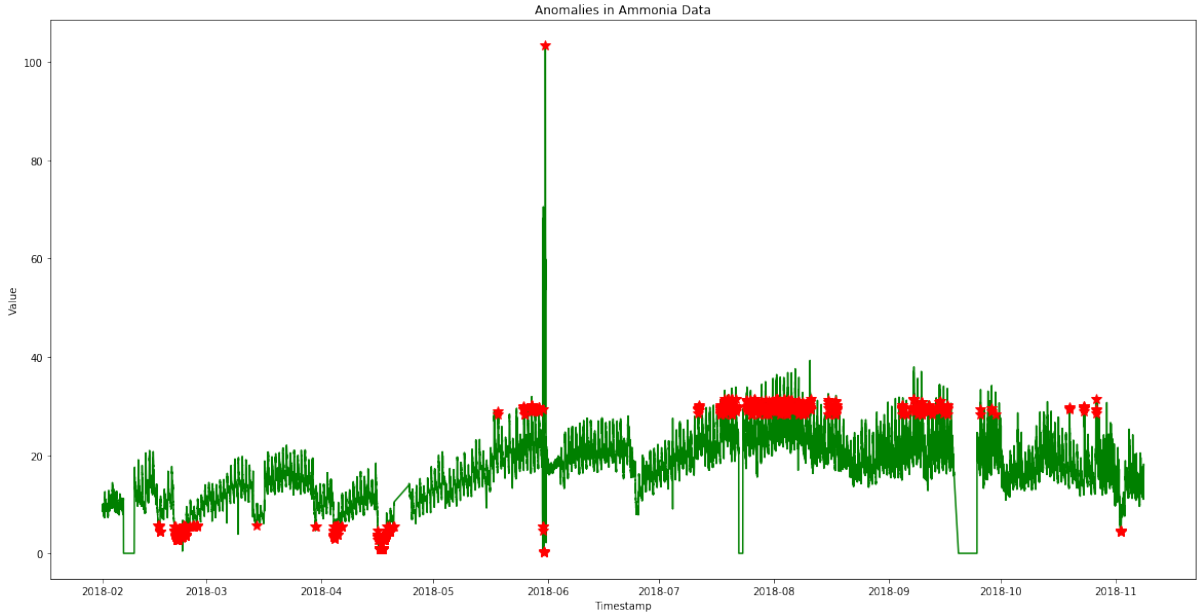


FIGURE 4.15: Chart of LOF Result for Ammonia Data 2018

Figure 4.16 illustrates the positions of all the anomaly instances detected by LOF in the chloride test data set.

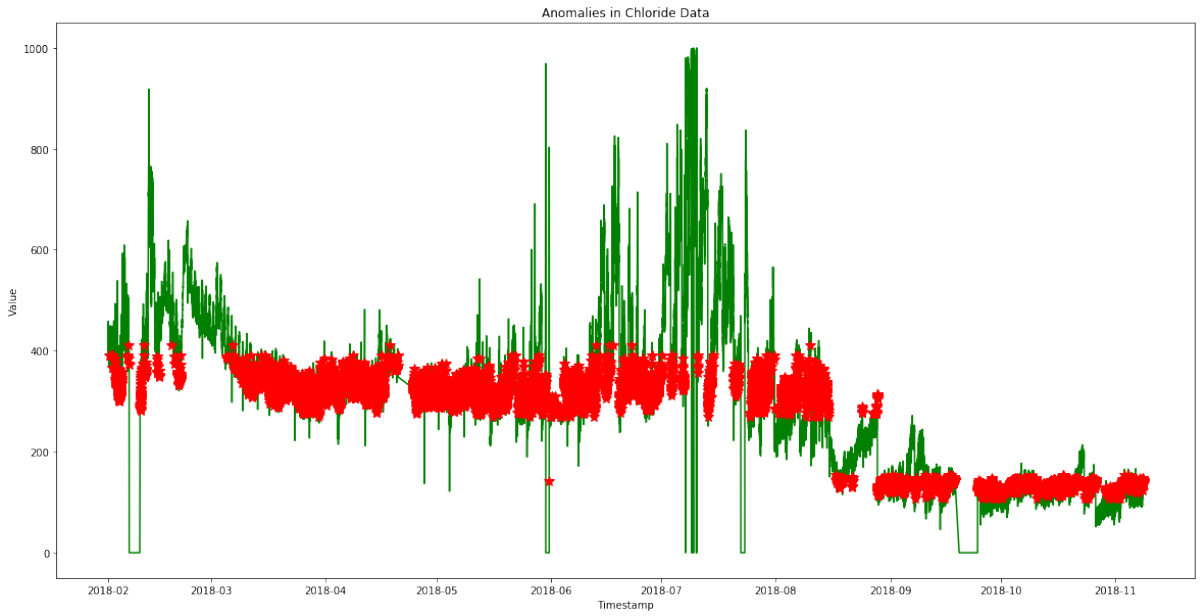


FIGURE 4.16: Chart of LOF Result for Chloride Data 2018

Figure 4.17 illustrates the positions of all the anomaly instances detected by LOF in the potassium test data set.

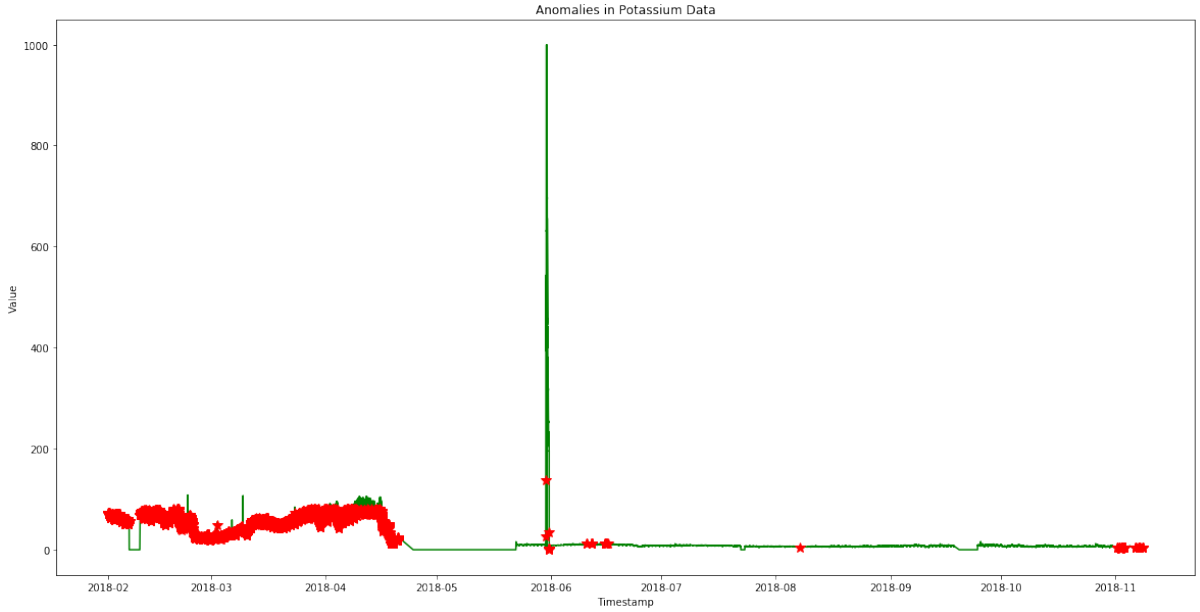


FIGURE 4.17: Chart of LOF Result for Potassium Data 2018

The result in Table 4.10 and the above figures indicate that LOF has found many anomalies whereas S-H-ESD and EMA only have detected a few of them. As discussed in Section 2.4.1, S-H-ESD is designed to detect seasonal time series, as well as underlying trends. But Figure 3.3, Figure 3.4, Figure 3.5 and Figure 3.6 indicate that none of the four data sets is either seasonal nor presents any trend. All three approaches do not work well if there are too many consecutive anomalies.

4.2.2 Execution Time

Table 4.11 shows the execution time for each data set. For a large data set with 395,715 records, LOF takes a few seconds to detect anomalies. Whereas, S-H-ESD needs two minutes for detection and EMA takes around one and half minutes for each data set.

Sensors	LOF	S-H-ESD	EMA
Temperature	5.81 s	2 min 9 s	1 min 30 s
Ammonia	5.38 s	1 min 55 s	1 min 32 s
Chloride	5.15 s	1 min 39 s	1 min 32 s
potassium	32.4 s	1 min 38 s	1 min 31 s

TABLE 4.11: Execution Time for Data of 2018

The result of second experiment confirms that LOF is much faster than S-H-ESD and EMA. Overall, all three methods are fast enough for real-time anomaly detection.

Chapter 5

Conclusions and Future Work

A comprehensive evaluation of five different anomaly detection algorithms on two data sets from a water sensor was performed in the primary experiment. In the second experiment, an additional four very large data sets with nearly 300,000 records were tested.

Local Outlier Factor successfully detects all 26 anomalies in the time series data in the primary experiment. The k-nearest neighbor local density based algorithms are not only simple to understand but easy to implement. More importantly, the non-parametric nature of algorithm makes it a genius algorithm for unsupervised learning. With zero to little training time, it could be a very powerful tool for anomaly detection of water quality data.

By taking seasonality and trend into accounts, S-H-ESD can detect both point as well as contextual anomalies in temporal data. Only some of outliers were detected mainly because there are missing values and noise data in the test data set. However, it is still a powerful and robust way to detect anomalies on seasonal time series data.

EMA can be useful if the start time and end time of the anomaly occurs are the concern.

The outlier-ness of IF is based on the threshold which is very difficult to tune to differentiate abnormal data from normal data. However, it can be used for validation. For example, the instance which is detected as anomaly by other algorithms can be safely classified as anomaly if it is also rated a high score (like 0.65) by IF.

RRCF marks some outliers with low anomaly scores (like the zeros) while marks some normal points with very high anomaly scores, which makes it impossible to filter anomalies out based on scores. This is because RRCF is very sensitive and even a slight deviation will result in high score. This problem might be reduced if the raw data is smoothed. The execution times was too long which is expected as it takes time to construct trees and compute depths and lengths. A powerful server would be required for real-time detection using RRCF.

LOF is much faster than S-H-ESD, EMA, IF and RRCF on detecting anomalies.

In the experiments, water temperature and ammonia data from the RSM30 system were used. Other sensor detected content will be verified once enough test data is collected.

Only short periods of test data were used for experiments in which all tests run against offline raw data. A procedure should be established to pre-process raw data in order to reduce noise by smoothing the test data set. For example, fill missing data by be mean of surrounding values and remove invalid values. Shortening the time interval of sensor data is another option

to smooth data and increase quality. A validation mechanism can be built to combine those algorithms together to generate more accurate results.

Bibliography

Aleskerov, E., Freisleben, B., and Rao, B. (1997). CARDWATCH: a neural network based database mining system for credit card fraud detection. In: *Proceedings of the IEEE/IAFE 1997 Computational Intelligence for Financial Engineering (CIFEr)*. IEEE, 220–226. DOI: 10.1109/CIFER.1997.618940.

Altman, N. S. (1992). An Introduction to Kernel and Nearest-Neighbor Nonparametric Regression. *The American Statistician* 46(3), 175–185. DOI: 10.1080/00031305.1992.10475879. eprint: https://www.tandfonline.com/doi/pdf/10.1080/00031305.1992.10475879. URL: https://www.tandfonline.com/doi/abs/10.1080/00031305.1992.10475879.

Amer, M. and Goldstein, M. (2012). Nearest-Neighbor and Clustering based Anomaly Detection Algorithms for RapidMiner. In: DOI: 10.5455/ijavms.141.

Angiulli, F. (2005). Fast Condensed Nearest Neighbor Rule. In: *Proceedings of the 22Nd International Conference on Machine Learning*. ICML '05. Bonn, Germany: ACM, 25–32. ISBN: 1-59593-180-5. DOI: 10.1145/1102351.1102355. URL: http://doi.acm.org/10.1145/1102351.1102355.

Arindam Banerjee, V. C. and Kumar, V. (2009). Anomaly detection : A survey. *ACM Computing Surveys* 41(3), 1–58.

Atkeson, C. G., Moore, A. W., and Schaal, S. (1997). Locally Weighted Learning. *Artif. Intell. Rev.* 11(1-5), 11–73. ISSN: 0269-2821. DOI: 10.1023/A:1006559212014. URL: https://doi.org/10.1023/A:1006559212014.

Auskalnis, J., Paulauskas, N., and Baskys, A. (2018). Application of Local Outlier Factor Algorithm to Detect Anomalies in Computer Network. *Elektronika ir Elektrotechnika* 24. DOI: 10.5755/j01.eie.24.3.20972.

Breunig, M. M., Kriegel, H.-P., Ng, R. T., and Sander, J. (2000). LOF: Identifying Density-based Local Outliers. *SIGMOD Rec.* 29(2), 93–104. ISSN: 0163-5808. DOI: 10.1145/335191.335388. URL: http://doi.acm.org/10.1145/335191.335388.

Chandola, V., Banerjee, A., and Kumar, V. (2009). Anomaly Detection: A Survey. *ACM Comput. Surv.* 41(3), 15:1–15:58. ISSN: 0360-0300. DOI: 10.1145/1541880.1541882. URL: http://doi.acm.org/10.1145/1541880.1541882.

Cleveland, R. B., Cleveland, W. S., McRae, J. E., and Terpenning, I. (1990). STL: A Seasonal-Trend Decomposition Procedure Based on Loess (with Discussion). *Journal of Official Statistics* 6, 3–73.

Cortes, C. and Vapnik, V. (1995). Support-vector networks. *Machine Learning* 20(3), 273–297. ISSN: 1573-0565. DOI: 10.1007/BF00994018. URL: https://doi.org/10.1007/BF00994018.

Ding, Z. and Fei, M. (2013). An Anomaly Detection Approach Based on Isolation Forest Algorithm for Streaming Data using Sliding Window. *IFAC Proceedings Volumes* 46(20). 3rd IFAC Conference on Intelligent Control and Automation Science ICONS 2013, 12–17. ISSN: 1474-6670. DOI: https://doi.org/10.3182/20130902-3-CN-3020.00044. URL: http://www.sciencedirect.com/science/article/pii/S1474667016314999.

Ertoz, L., Lazarevic, A., E. Eilertson, L. A., Tan, P., Dokas, P., Kumar, V., and Srivastava, J. (2003). Protecting Against Cyber Threats in Networked Information Systems. *SPIE Annual Symposium on AeroSense, Battlespace Digitization and Network Centric Systems III*.

Fomel, S. and Claerbout, J. F. (2009). Guest Editors' Introduction: Reproducible Research. *Computing in Science Engineering* 11(1), 5–7. ISSN: 1521-9615. DOI: 10.1109/MCSE.2009.14.

Garcia, S., Derrac, J., Cano, J., and Herrera, F. (2012). Prototype Selection for Nearest Neighbor Classification: Taxonomy and Empirical Study. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 34(3), 417–435. ISSN: 0162-8828. DOI: 10.1109/TPAMI.2011.142.

Grubbs, F. E. (1969). Procedures for Detecting Outlying Observations in Samples. *Technometrics* 11(1), 1–21. DOI: 10.1080/00401706.1969.10490657. eprint: https://www.tandfonline.com/doi/pdf/10.1080/00401706.1969.10490657. URL: https://www.tandfonline.com/doi/abs/10.1080/00401706.1969.10490657.

Guha, S., Mishra, N., Roy, G., and Schrijvers, O. (2016). Robust Random Cut Forest Based Anomaly Detection on Streams. In: *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*. ICML'16. New York, NY, USA: JMLR.org, 2712–2721. URL: http://dl.acm.org/citation.cfm?id=3045390.3045676.

Hamming, R. W. (1950). Error detecting and error correcting codes. *The Bell System Technical Journal* 29(2), 147–160. ISSN: 0005-8580. DOI: 10.1002/j.1538-7305.1950.tb00463.x.

Hariri, S., Carrasco Kind, M., and Brunner, R. J. (2018). Extended Isolation Forest. *CoRR* abs/1811.02141.

He, Z., Xu, X., and Deng, S. (2003). Discovering cluster-based local outliers. *Pattern Recognition Letters* 24(9-10), 1641–1650.

Ho, T. K. (1995). Random Decision Forests. In: *Proceedings of the Third International Conference on Document Analysis and Recognition (Volume 1) - Volume 1*. ICDAR '95. Washington, DC, USA: IEEE Computer Society, 278–. ISBN: 0-8186-7128-9. URL: http://dl.acm.org/citation.cfm?id=844379.844681.

Hochenbaum, J., Vallis, O. S., and Kejariwal, A. (2017). Automatic Anomaly Detection in the Cloud Via Statistical Learning. *CoRR* abs/1704.07706.

Jamie Bartram World Health Organization, U. N. E. P. and Ballance, R. (1996). Water quality monitoring : a practical guide to the design and implementation of freshwater quality studies and monitoring programs / edited by Jamie Bartram and Richard Ballance. *Computing in Science Engineering*. URL: http://www.who.int/iris/handle/10665/41851.

Jie Wang Jia Liu, L. W. and Xu, Y. (2019). An Introduction to Kernel and Nearest-Neighbor Nonparametric Regression. *Artificial Intelligence*, 1–12. ISSN: 0941-0643. DOI: https://doi.org/10.1007/s00521-019-04066-3.

Jin, W., Tung, A. K. H., Han, J., and Wang, W. (2006). Ranking Outliers Using Symmetric Neighborhood Relationship. In: *Advances in Knowledge Discovery and Data Mining*. Ed. by W.-K. Ng, M. Kitsuregawa, J. Li, and K. Chang. Berlin, Heidelberg: Springer Berlin Heidelberg, 577–593. ISBN: 978-3-540-33207-7.

Jupyter (2019). Jupyter Notebook. https://jupyter.org. Jupyter Notebook. URL: https://jupyter.org/.

Kriegel, H.-P., Kröger, P., Schubert, E., and Zimek, A. (2009). LoOP: Local Outlier Probabilities. In: *Proceedings of the 18th ACM Conference on Information and Knowledge Management*. CIKM '09. Hong Kong, China: ACM, 1649–1652. ISBN: 978-1-60558-512-3. DOI: 10.1145/1645953.1646195. URL: http://doi.acm.org/10.1145/1645953.1646195.

LinkedIn (2015). Anomaly Detection and Correlation library. https://github.com/linkedin/luminol. Luminol is a light weight python library for time series data analysis.The two major functionalities it supports are anomaly detection and correlation.

Liou, C.-Y., Cheng, W.-C., Liou, J.-W., and Liou, D.-R. (2014). Autoencoder for Words. *Neurocomput.* 139, 84–96. ISSN: 0925-2312. DOI: 10.1016/j.neucom.2013.09.055. URL: http://dx.doi.org/10.1016/j.neucom.2013.09.055.

Liu, F. T., Ting, K. M., and Zhou, Z.-H. (2012). Isolation-Based Anomaly Detection. *TKDD* 6, 3:1–3:39.

Marcnuth (2018). Twitter's Anomaly Detection in Pure Python. https://github.com/Marcnuth/AnomalyDetection. This repository aims for rewriting twitter's Anomaly Detection algorithms in Python, and providing same functions for user.

McCloskey, M. and Cohen, N. J. (1989). Catastrophic Interference in Connectionist Networks: The Sequential Learning Problem. In: ed. by G. H. Bower. Vol. 24. Psychology of Learning and Motivation. Academic Press, 109–165. DOI: https://doi.org/10.1016/S0079-7421(08)60536-8. URL: http://www.sciencedirect.com/science/article/pii/S0079742108605368.

Merigo, J. M. and Casanovas, M. (2012). A New Minkowski Distance Based on Induced Aggregation Operators. *International Journal of Computational Intelligence Systems* April 2011, 123–133. DOI: 10.1080/18756891.2011.9727769.

Olivier Chapelle Bernhard Schölkopf, A. Z. (2006). Semi-Supervised Learning.

Pack Kaelbling, L., Littman, M., and P Moore, A. (1996). Reinforcement Learning: A Survey. *Journal of Artificial Intelligence Research* 4, 237–285.

Primodal (2013). RSM30 Getting Started Guide. 23. Primodal System Inc., 8.

Ron Kohavi, F. P. (1998). *Machine Learning*. Kluwer Academic Publishers, 271–274.

Rosner, B. (1983). Percentage Points for a Generalized ESD Many-Outlier Procedure. *Technometrics* 25(2), 165–172. ISSN: 00401706. URL: http://www.jstor.org/stable/1268549.

Samuel, A. (1959). Some Studies in Machine Learning Using the Game of Checkers. *IBM Journal of Research and Development* 3, 210–229. DOI: 10.1147/rd.33.0210.

Song, X., Wu, M., Jermaine, C., and Ranka, S. (2007). Conditional Anomaly Detection. *IEEE Transactions on Knowledge and Data Engineering* 19(5), 631–645. ISSN: 1041-4347. DOI: 10.1109/TKDE.2007.1009.

Spence, C., Parra, L., and Sajda, P. (2001). Detection, synthesis and compression in mammographic image analysis with a hierarchical image probability model. In: IEEE, 3–10. DOI: 10.1109/MMBIA.2001.991693.

Stigler, S. M. (1986). *The history of statistics - The measurement of uncertainty before 1900.*

Stuart J. Russell, P. N. (2009). *Artificial Intelligence: A Modern Approach*. third. Prentice Hall.

Tang, J., Chen, Z., Fu, A. W.-c., and Cheung, D. W. (2002). Enhancing Effectiveness of Outlier Detections for Low Density Patterns. In: *Advances in Knowledge Discovery and Data Mining*. Ed. by M.-S. Chen, P. S. Yu, and B. Liu. Berlin, Heidelberg: Springer Berlin Heidelberg, 535–548. ISBN: 978-3-540-47887-4.

Twitter (2015). Anomaly Detection. https://github.com/twitter/AnomalyDetection. AnomalyDetection R package.

Wenzel, F., Galy-Fajou, T., Deutsch, M., and Kloft, M. (2017). Bayesian Nonlinear Support Vector Machines for Big Data. In: *Machine Learning and Knowledge Discovery in Databases*. Ed. by M. Ceci, J. Hollmén, L. Todorovski, C. Vens, and S. Džeroski. Cham: Springer International Publishing, 307–322. ISBN: 978-3-319-71249-9.

Wilcox, C., Woon, W. L., and Aung, Z. (2013). *Applications of machine learning in environmental engineering*. Tech. rep.