Micro-Service System Troubleshooting

Henry Zheng Sahand Sabour Samuel Pegg

Abstract

With the rapid growth of large micro-service based systems, detecting anomalies in system operations has become a significantly essential task. In addition, the root causes of such anomalies must also be detected to prevent major costs and losses. Due to the large number of existing calls and complex relationships between different micro-services in modern systems, manual inspection of such micro-services has become extremely challenging and rather impossible. Hence, developing a set of methods that accomplish these tasks within a timely manner and with high accuracy is highly necessary. In this report, we propose an online algorithm that analyzes data generated from a micro-service based system and detects the occurring anomalies and their corresponding root causes.

Keywords: micro-service, anomaly detection, root cause analysis, root cause detection

1 Introduction

With the recent trends, increasing number of modern day large-scale systems are utilizing micro-services in their system architecture design. In a micro-service based architecture, the system operations are provided as different services, which are loosely coupled (i.e. independent from each other), organized, and highly maintainable, analyzable, and testable. Hence, this type of architecture allows large-scale and complex systems to provide faster and more reliable services to their users. However, a major disadvantage of these systems is the massive number of calls between different services and their complex relationships; which in turn makes finding faulty services and errors rather challenging.
In the case that a faulty micro-service causes an error in the system, this occurrence would be referred to as an anomaly, which should be detected as soon as possible. Accordingly, the anomaly should be traced back to the faulty service that caused it. This process is referred to as troubleshooting and by accomplishing this task, the system maintainers would be able to fix the faulty service and any subsequent problems before experiencing major costs and losses. Therefore, implementing an algorithm that detects system's anomalous behavior and is able to both accurately and efficiently identify its root cause is essential.
In this project, we were tasked to design an online algorithm to perform these tasks on an incrementally published dataset (i.e. real-time data) generated by a micro-service based system through Kafka producer. More specifically, our task was to analyze the provided time series data, detect anomalous behavior, discover anomalous system nodes at the time

of this occurrence, and find the Key Performance Indicator (KPI) that is the root cause of this anomaly. The rest of this report is organized as follows: in section 2, we summarize the relative literature for these tasks and acknowledge the work of top teams for this competition; in section 3, we further explain the problem statement and provide our implemented algorithms for this project; section 4 includes a discussion of our results as well as the lessons learned; lastly, we conclude the report in section 5.

2 Related Work

As mentioned, micro-service based large-scale systems consists of large number of complex calls and relationships between different services. In addition, once a service in the system becomes unavailable, unstable, and/or impaired, detecting and fixing this system in a timely and accurate manner is the top priority. Due to the large amount of data to be analyzed, manual inspection has proven to be hugely time consuming and overall, an ineffective approach. Hence, considerable research has been conducted to automatically analyze and detect such failures and their causing factors.
For the task of anomaly detection, the problem statement would be fairly simple: given the history of seasonal KPI data from a micro-service based system, create a time series and detect the present outliers. Donut [1] and Bagel [2] managed to accomplish this task by utilizing variational auto-encoders (VAEs). However, these models achieve satisfactory results for this task; however, they need sufficient resources for training and given that we were not provided with any GPUs, they were not suitable for our project. In addition, TraceAnomaly [3], a deep Bayesian network based algorithm, also managed to obtain accurate results within a short duration. However, given that it is a deep learning algorithm, similar to the previous two approaches, it was not suitable for our project.
For our method, we drew huge inspiration from MicroRCA [4] as the proposed framework and their studied cases strongly resembled this competition. Their approach would first detect anomalies in the time series data using BIRCH [5]. As will be mentioned later in section 3, we also used this algorithm for anomaly detection in the first part of our approach. Accordingly, an attributed graph between the hosts and services are created and the subgraph corresponding to the detected anomaly is extracted. Lastly, the edges and nodes of the extracted sub-graph are weighted using Pearson's correlation function, and the faulty services are localized by utilizing Personalized PageRank [6]. We implemented different variations of weighting and PageRank but were unable to produce satisfactory results.
Regarding the proposed algorithms for this competition, the winning team [7] cleverly discarded one of the three provided data sources to increase the detection speed. Accordingly, they established a set of rules and manually selected thresholds to detect anomalies and localize root causes. Similarly, the runner up team [8] also discarded the first set of provided KPI sources, focused merely on trace data and formulated this problem as a pattern finding challenge in a constructed anomaly table. Lastly, the team in third place [3] utilized the first set of available data to identify the anomalous points, analyzed the second set to detect failed and delayed calls, find abnormal hosts, and extract the necessary features. One of the honorable mentions in their work was that they used dictionaries instead of pandas to improve the algorithm's speed.

3 Methodology

In this section, we describe the problem statement and our implemented algorithms.

3.1 Problem Statement

In this project, we were provided with three KPI data sources: ESB, Trace, and Host data.

ESB business indicator (ESB): it is provided every minute and mainly demonstrates the number of requests for the *osb_001* service and the overall success rate of these request during each minute. Once an anomaly occurs, assuming at a point t in time, the success rate is expected to be lower than 1. Accordingly, t would be recorded to be used for further analysis in the other two KPI data sources.

Trace: it is provided for every request and consists of several micro-service calls, referred to as spans. This section of the data demonstrates the start and elapsed time for each span, the databases that were accessed (if any), the trace of the span and its host. Upon finding anomalous time t in the ESB data, the time around t is to be investigated in order to detect anomalous spans (i.e. nodes with unusually long response time) and ultimately, realize faulty service nodes.

Host: are provided in the (timestamp, value) format and includes the host service name and the called operation. Upon finding the faulty service nodes, this data can be explored to find anomalous values for a KPI, which would then be flagged as a root cause.

3.2 ESB Business Indicator

Inspired by [4], we utilize BIRCH [5], which is an online clustering-based outlier detection algorithm, to detect anomalies in the ESB data. In this project, we considered comparatively large values for average time and considerably small success rates as anomalous behavior. Hence, we employed BIRCH separately for these two columns and set the BIRCH threshold (i.e. radius) to 0.5 and 0.1 respectively.
The algorithm produced good results for a short while. However, a major disadvantage of this implementation was that when the code was left running on the server, the anomaly detection process started happening when the system is in its normal state. Given enough anomalies, the anomalies actually become the normal state of the system. In order to address this problem, we pretrained two separate BIRCH models for average time and success rate respectively on the provided two weeks of data with all anomalous data removed.
Upon detecting an anomaly in the ESB data, we would stop analyzing incoming data until a root cause is found. Accordingly, we would send the timestamp of the detected anomaly to the next module for faulty service detection in trace data. The incoming data would not be discarded but rather stored in a separate storage for future analysis.
However, this approach also proved to be sub-optimal. The reason being, the effects of many anomalies are often quite significantly delayed in the ESB data - indeed, some anomalies do not cause any change in the ESB data. Thus waiting on the ESB data for anomalies was inefficient and sometimes causing us to miss anomalies. Hence we decided to remove this part of the project.

3.3 Final Solution Design

The diagram below shows an overview of the method.

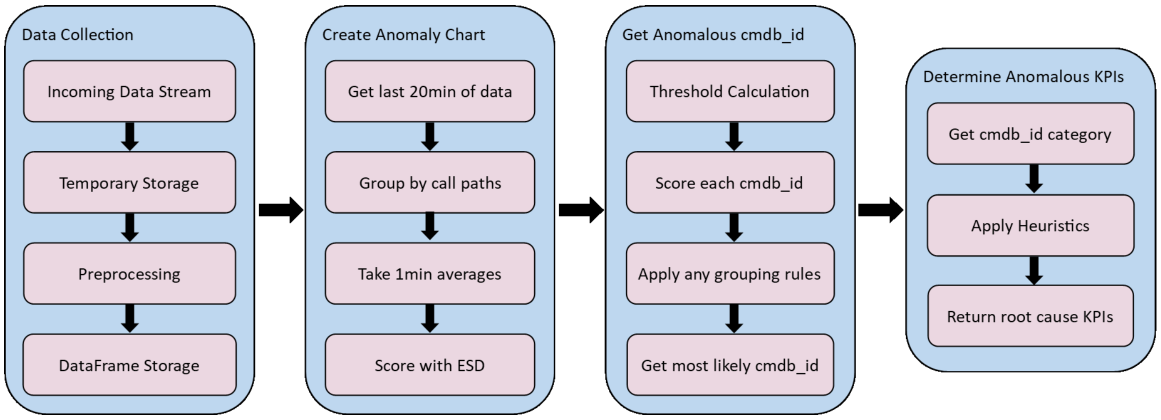

FIGURE 1: Proposed method architecture

3.3.1 Data Collection

3.3.2 Anomaly Chart

3.3.3 Chart Analysis

Threshold Calculation: We firstly calculate a threshold t_1 for the anomaly chart. We found that 20% of the maximum value in the table, M, was a good indicator for anomalous entries. We also set a minimum threshold of 10 so that in the case of a normal table, no anomalies are detected. Hence,

$$t_1 = \max(0.2M, 10)$$

cmdb_id Scores: For each cmbd_id, i, we calculate the mean of the row sum and the column sum (when the column exists). Call the resulting value m_i. Then, using the threshold, for each cmbd_id, we also calculate the number of anomalous entries in its row and the number of anomalous entries in its column. Let r_i be the number of anomalies in the row, and c_i the number of anomalies in the column. We then calculate

$$a_i = \left\lfloor \frac{2r_i + c_i}{2} \right\rfloor$$

To assign a score to each cmdb_id, we simply calculate $s_i = a_i \times m_i$. We use another threshold

$$t_2 = \max_i(0.1 * s_i, 1)$$

to filter out the cmdb_ids with low score (s_i) and pass the resulting list on to the next stage. If there are no scores s_i greater than 1, we return the empty list and complete localization.
Apply any grouping rules: The rules we apply are as follows:

- If os_021 and os_022 are anomalous, the problematic cmdb_id is os_001.

- If we have docker_00(x) and docker_00($x + 4$), the problematic cmdb_id is actually os_0($16 + x$). For example, if docker_002 and docker_006 are anomalous, the problematic cmdb_id is os_018.
- We have a “fly remote" row in our anomaly chart. The cmdb_id corresponding to an anomalous fly remote is os_009.

Get most likely cmdb_id: If none of the combination rules are met, we simply send the cmdb_id with the greatest score to the next stage. Else, we send the successful result of one of the combination rules. This means we prioritise the grouping rules over score, as this gave us the best performance.

3.3.4 KPI Localisation

Given the anomalous cmdb_id from the previous stage, we use the following observations to determine the root cause KPIs.

- If the cmdb_id is of type “os", i.e. an operating system, the KPIs are always Sent_queue and Received_queue.
- If the cmdb_id is of type “docker", the KPI root cause is either Null or container_cpu_used.
- If the cmdb_id is of type “db", i.e. a database, the KPI root causes are either the combination of Sess_Connect, Proc_Used_Pct and Proc_User_Used_Pct, or the combination of On_Off_State and tnsping_result_time.

So if the cmdb_id is of type “os" we can instantly send off a result, but docker and db cmdb_ids require further analysis to decide between the two options. Lets consider the docker anomalies first.
If the docker-docker entry in the anomaly chart (i.e. a call to itself) is anomalous, the problem is container_cpu_used. If the self call is not anomalous, then it must be a network issue, in which case the KPI is Null.
Moving on to the database case, for to decide between the two options we need only look at the On_Off_State KPI. If there are any 0 values in On_Off_State, we return On_Off_State and tnsping_result_time. If the values for On_Off_State are all 1, we return Sess_Connect, Proc_Used_Pct and Proc_User_Used_Pct.

3.4 Additional Checks

Since we run anomaly detection every minute, and we take 1 minute averages of the last 20 minutes of data, sometimes an anomaly is not fully expressed in the data during the first minute of the anomaly. This means that if we blindly send off a result as soon as we get a table, we will often send the wrong answer. To solve the issue, when we get a result from the table we store it and run anomaly detection a minute later. Only if the two answers are equal do we send off the result to the server. Otherwise, we keep running anomaly detection every minute until the last two results agree. From our observations of the code on the server, the first result is usually correct, and the next result is the same, so we can detect anomalies in two minutes. However, sometimes, particularly for docker anomalies, the database values spike slightly above the threshold of 10 before the docker entries in the table become anomalous. Originally we would have submitted a database anomaly, but now we are correctly able to identify the actual root cause - the docker.

4 Discussion and Lessons Learned

In the final test of this competition, we were able to score 70 points and rank 8th. After finishing this competition and analyzing the proposed algorithms of the top teams, we believe that we learned a number of valuable lessons throughout this experience: first, in anomaly detection, data processing is the most important step of the method and should not be undermined. With efficient data processing, the important features of the available data, which is highly beneficial for the proceeding steps of the algorithm, could be extracted; second, during this competition, we had the opportunity to work with seasonal data and different deep learning algorithms and approaches, which served as a great learning experience; third, judging by the methods that scored the most points, we learned that over complicating the problem is not a good idea and we should always try to make easy solutions work first; last, the most valuable lesson of all for us was to always make sure that all parts of the program are able to work as expected. For instance, in the final days of testing, we forgot to uncomment the submit function, which resulted in our team to score 0 points regardless of our improvements in the code. Hence, to conclude, the overall lesson would be to never take parts of your work for granted and ensure all parts are working as planned, try the simple approaches first, and learn as you work.

5 Conclusion

In conclusion, we proposed a rather robust online algorithm for anomaly detection and root cause localization in micro-service based systems. We introduced the related work in the literature and the proposed method of fellow competitors. In addition, we demonstrated our proposed methodology and discussed the lessons learned throughout this project. Regardless of the final results, we regard this competition as a valuable learning experience and are proud of our team's hard work to obtain these results.

References

[1] Haowen Xu, Wenxiao Chen, Nengwen Zhao, Zeyan Li, Jiahao Bu, Zhihan Li, Ying Liu, Youjian Zhao,Dan Pei, Yang Feng. "Unsupervised Anomaly Detection via Variational Auto-Encoder for Seasonal KPIs in Web Applications". Proceedings of the 2018 World Wide Web Conference on World Wide Web, 2018, pp. 187-196.

[2] Zeyan Li, Wenxiao Chen, Dan Pei. "Robust and Unsupervised KPI Anomaly Detection Based on Conditional Variational Autoencoder". 2018 IEEE 37th International Performance Computing and Communications Conference (IPCCC). IEEE, 2018.

[3] Ping Liu, Haowen Xu, Qianyu Ouyang, Rui Jiao, Zhekang Chen, Shenglin Zhang, Jiahai Yang, Linlin Mo, Jice Zeng, Wenman Xue, Dan Pei. "Unsupervised Detection of Microservice Trace Anomalies through Service-Level Deep Bayesian Networks". 31th International Symposium on Software Reliability Engineering (ISSRE). IEEE, 2020.

[4] Li Wu, Johan Tordsson, Erik Elmroth, Odej Kao. "MicroRCA: Root Cause Localization of Performance Issues in Microservices". IEEE/IFIP Network Operations and Management Symposium (NOMS), 2020.

[5] A. Gulenko, F. Schmidt, A. Acker, M. Wallschlager, O. Kao, and F. Liu, “Detecting anomalous behavior of black-box services modeled with distance-based online clustering”, in 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), 2018.

[6] G. Jeh and J. Widom, “Scaling personalized web search”, in WWW-2003, pp. 271–279.

[7] Yunfeng Zhao, Xuanrun Wang, Guojie Fan. "Advanced network management - the Old Driver on Xuetang Road", 2020.

[8] Yixiong Ji, Yunpeng Liu. "Anomaly Detection and Root Cause Localization in Microservice System - meow meow group", 2020.

[9] Zhangzi Hao, Yaoyu Zhang, Shaohuai Liu. "ANM project - Veritaserum", 2020.