MACHINE LEARNING FOR LOG DATA ANALYSIS

LOG DATA ANALYSIS FOR SOFTWARE DIAGNOSIS: THE MACHINE LEARNING THEORIES AND APPLICATIONS

By

YIXIN HUANGFU, M.Sc., B.Sc.

A Thesis Submitted to the School of Graduate Studies

in Partial Fulfillment of the Requirements

for the Degree of Doctor of Philosophy

in the Department of Mechanical Engineering

McMaster University, Hamilton, ON, Canada

DOCTOR OF PHILOSOPHY (2021), McMaster University, Hamilton, Ontario, Canada

(Mechanical Engineering)

TITLE: Log Data Analysis for Software Diagnosis: The Machine Learning Theories and Applications

AUTHOR: Yixin Huangfu, M.Sc., B.Sc.

SUPERVISOR: Dr. Saeid Habibi

Number of Pages: xx, 217

Abstract

This research investigates software failure and fault analysis through data-driven machine learning approaches. Faults can happen in any software system and may hugely impact system reliability and user experience. Log data, the machine-generated data that records the system status, is often the primary source of information to track down a fault. This study aims to develop automated systems that recognize recurring faults by analyzing the system log data. The methodology developed in this research applies to the Ford SYNC vehicle infotainment system as well as other systems that produce similar log data.

Log data has been used in manual examination to help trace and localize a fault. This manual process can be effective and sometimes the only feasible way of troubleshooting software faults. However, as the amount of log data increases significantly with the growing complexity and scale of software, the manual workload can get overwhelming. During the system-level validation tests, all system components are producing log data, resulting in tens of thousands of lines of log messages in just a few minutes. Therefore, automated diagnosis has been a promising approach for log data analysis.

Three machine learning approaches are investigated in this research to tackle the fault diagnosis problem: 1) the data mining approach; 2) the statistical feature approach; and, 3) the deep learning approach. The first method attempts to mimic human experts to examine log data. Log sequences representing a fault are extracted through data mining techniques and used to identify anomalies. The method is effective when applied to a small volume of data, but computational efficiency can be an issue when scaling to larger datasets. As its name suggests, the second method involves an examination of the log data’s statistical and numerical features and adapting a machine learning model for decision making. The use of numerical features to describe log data has significant computational efficiency improvement over working directly with sequences. The last approach adopts deep learning models that process the log data in sequential format, enabling more sophisticated feature extraction that often exceeds human capability. In this research, all three methods are implemented and evaluated in a controlled testing environment, and their strengths and weaknesses are comparatively evaluated.

This study also reports on a novel finding that the time information in a log sequence plays an important role in distinguishing a faulty condition from a normal one. For most software systems, the log sequences are unevenly spaced, meaning that the timestamps associated with log data are nonuniform. Existing log analysis studies generally overlooked the time information while emphasizing log sequences. This research proposes a novel deep learning structure to unify the processing of timestamps and log sequences. The timestamps are integrated through interpolation at an intermediate layer of a neural network. Testing results demonstrate that the inclusion of timestamps makes a significant contribution to identifying a fault, and that models using time stamps can push the performance to a higher level.

Acknowledgements

First and foremost, I would like to express my greatest gratitude to my supervisor, Dr. Saeid Habibi. Without his support and guidance, my Ph.D. journey would not be a reality. Dr. Habibi demonstrates what a good supervisor should be, patient, righteous, and considerate. His role modelling has inspired my research interests and made me believe in a career in academia. I also learned how to teach, communicate with, and collaborate with fellow students and colleagues in these fruitful years. These influences will last for the rest of my life. And I look forward to working with you in the future.

I would also like to acknowledge Mr. Cam Fisher, who took care of the logistics and helped me focus on my research. He has incredible industrial experience and is always the go-to guy whenever I have a problem with the hardware. My sincere gratitude also goes to my committee members, Dr. Alan Wassyng and Dr. Steven Veldhuis, for their sharp insights and constructive feedback.

To my fellow students in the lab, I miss the days when we are physically in the lab, the pizza lunches, the casual talks, and the technical discussions. You are the first friends I made in Canada and painted a diverse picture of McMaster University. I was impressed and inspired by your unique stories and mindsets. My mind has never been so open before.

My journey would not commence without the support from my wife. You gave me the courage to change the status quo. My appreciation also goes to my parents, who supported my decision to leave a well-paid job and explore possibilities in a new world.

Last but not least, I would like to thank McMaster University and the Department of Mechanical Engineering. I am proud to be one of your graduates.

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

Data analytics has been receiving increasing attention from both industry and academia for the past decade. It has become one of the most important emerging areas of interest in various fields, including finance, computing and software, and manufacturing. The rise of this field is mainly due to people and organizations realizing the value of extracting the information contained in massive amounts of data, which in turn stimulates extensive data collection. For example, a recent survey on corporate finance found that companies with comprehensive analytics are more likely to make effective decisions and achieve top financial performance [1]. The benefits of information buried in data are not limited to finance and marketing. As new tools and methodologies are being developed for data analytics, new areas of interest start to emerge in both industry and academia.

The major challenges to transform the data into financial benefits often come from the sheer volume of data to be analyzed. Based on a survey in 2018, the volume of data created, captured, copied, and consumed worldwide is at the zettabyte level, i.e. 10^{21} bytes, and is ever increasing [2]. The vague term Big Data is widely known to describe this phenomenon. The newly generated data mainly comes from the digitalization of existing non-digital information, such as books, and sensor-enabled data-collecting devices, such as in autonomous vehicles [3]. Although human beings are exceptionally capable of data comprehension, many modern datasets are too large to be analyzed by humans manually. Computers are widely adopted to assist with the analysis, but often they still remain ineffective in processing large or complex data because of the way data is gathered. These emerging opportunities of Big Data are the driving force of data analysis research. The newly developed methods rely on Artificial Intelligence (AI) and bring a new perspective to problem solving.

The analysis of digital images is a good example of the changes brought by the large volume of data. Recognizing objects from an image has been a common task in computer vision research. It is a simple task for human beings, while automation using computer programs is a difficult task. Systematic studies have been emphasizing the extraction of distinct features that are often explicitly specified, such as using staircase patterns to identify buildings in a satellite image [4]. Although the progressive achievements accumulated over decades has been comprehensive, it is the very recently developed deep convolutional neural networks [5] that completely reshaped the research. This new subset of machine learning methods is specifically designed to utilize an extremely large amount of data and learn implicit feature patterns from them. Based on this concept, various models have pushed computer-object-recognition to human-level performance and effectively made this technology commercially viable. Such innovation and progress will not happen without the abundance of data and data-driven methodologies being developed.

The development of AI-based data analytics is often related to specific domains. In the above example, it is for image processing, but for other areas different strategies and methods are needed. The various data types, although appearing differently, can be broken down into four fundamental categories: continuous, discrete, ordinal, and nominal, as shown in Figure 1.1.1. Continuous and discrete data are represented in a numerical form, such as the temperature of a day and the colour values of an image pixel. The ordinal and nominal data are represented in categories, such as the days in a week and the words from a written language. Packing the basic data types in a structured form constitutes complex data types. For example, the digital colour images are represented by a 3D matrix of discrete integer values. A vector is another common construction that appears in many applications. In electromechanical systems, a voltage sensor produces numerical data points in discrete time intervals that is known as time series. In natural language processing, a piece of text can be viewed as a sequence of words and punctuations which is a nominal type of data.

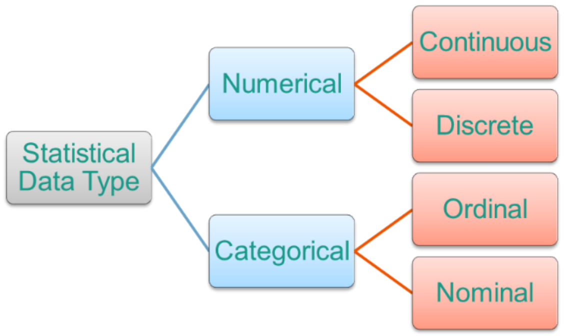


Figure 1.1.1 Statistical data types.

The structured data contains meaningful information about the system that produces them. The time series of sensor measurements can tell the operational state of an engine;

the text sequence reveals an author's sentiment when writing an article. The task of recognizing the underlying features and associating them to a category is often referred to as classification. It has a wide range of applications including fault diagnosis, object recognition, fraud detection and social media fact-checking.

This thesis will focus on the classification task through the analysis of categorical sequence data. Methodologies involving pattern searching, feature characterization, and classification models will be explored and discussed. The study originates from a project of analyzing system logs from a vehicle infotainment software, while the methodologies developed in this research can potentially be extended or adapted to other systems and applications as well.

1.1 Research Objectives

This research mainly focuses on investigating the log data from the Ford SYNC system for fault diagnosis purposes. The SYNC is an operating system running on embedded platforms. It features entertainment and communication functionalities, such as hands-free telephone calls, audio control, and air conditioning. As such systems evolve rapidly to include new functionalities, such as WiFi and cellular network connectivity and driver assistance technologies, the software's size and complexity continues to grow. Developing and maintaining the SYNC system requires collaborative work from a large team and an integrated testing process.

Testing is an essential aspect of software development to ensure quality by verifying that the product meets design requirements. From a low to a high level, these include unit testing, integration testing, system testing, and acceptance testing. This study is concerned with acceptance testing, which comes at the very last stage of development. In the Ford SYNC development scenario, the acceptance testing is performed with the whole system functionally ready on a hardware test bench or mule vehicle. All software components are individually tested through the lower level testing and most faults are resolved. However, even if software configuration items pass unit tests, it does not ensure the functionality of the whole system as system level faults may occur due to incomplete or missing software requirements. Often system level functional faults are only discovered during development or acceptance testing, where they can propagate to failures.

When unexpected software behaviour is found during acceptance testing, the first task is to synthesize and localize the fault. This is no easy task, especially for large-scale software that requires a whole team of developers to maintain. In order to trace the origin of a fault, the software under test is instrumented with diagnostics features that record its status during tests. A type of machine-generated form of condition monitoring data is called “system logs”. In software condition monitoring, examination of the system logs is the first step to identifying potential faults.

Manually examination of the system logs is a time-consuming process because of the large volume of log data that is normally generated continuously for condition monitoring. This generates a large workload for the software testing team. In particular, many reported anomalies turn out to be caused by the same fault, but they still require manual inspection of the logs. The objective of this research is to address recognizing the reoccurring faults in an automated manner. Specifically, a fault identification system is expected to:

- identify reoccurring faults that are conventionally being identified manually;
- perform the diagnosis mainly through analyzing system logs as the main source of data; and,
- perform the task without the availability of source code.

Note that the intention is not to perform a complete end-to-end diagnosis, but rather to speed up the process by screening the reported anomalies and finding out the reoccurring ones. Faults that have not been previously manually diagnosed still need attention from human experts. The source code is generally not available for the use case of this application, so one of the assumptions is that the software logging instrumentation is able to trace the propagation from a fault to a failure. This is reasonable because if the assumption does not hold, even a manual process could not locate the fault. The detailed introduction of the acceptance testing at Ford and the use case of an automated diagnosis system is provided in Chapter 2.

1.2 Analysis Approaches

Fault diagnosis methods in general include two categories: model-based and signal-based approaches. The model-based approach requires a system model to be available. A deviation in the system response compared to the expected output generated by the model would signal a fault condition and is used for fault detection and diagnosis. In software systems, the models can be constructed directly through the analysis of source code (the static analysis) or by tracing the output of software execution while given a known test input (the dynamic analysis). Both methods require the availability of source code.

The fault localization problem as presented by Yousefi [6] is an example of the system modelling approach. In the study, the test cases and test oracles – the mechanism of determining whether a test output is correct – are all known. The source code is instrumented such that each method in the source code produces an explicit tracing message during execution. By analyzing the tracing that pinpoints a certain block of code in the program, the method is able to create an execution tree model for a software test case. When a test produces a faulty output, the model is able to locate the branch that deviates from regular execution.

Alternatively, the signal-based approach attempts to model the system behaviour through analysis of the system output data without modelling how the system works internally. A self-learning system attempts to generalize distinguishing features from the data and gives a classification output. This method often requires a large amount of data and manual feature engineering effort in order to extract meaningful information from the data. It can work without the availability of source code, so it has different use cases than the model-based approach.

In the Ford system log analysis case, the task of recognizing reoccurring faults has a different setting from a typical software diagnosis application. At the initial stage of bug inspection (referred to as the *triage* process as later introduced in Chapter 2), the source code is unavailable or impractical to decipher due to its large size. Meanwhile, the system log trace is abundant and comprehensive, such that they can be understood by human testers. As a result, the signal-based approach is more preferred.

In this study, multiple machine learning approaches are investigated to perform fault identification through analyzing the system log data. Note that the term machine learning has different definitions. In general, it refers to the algorithms that can self-adjust their parameters to fit the given data (signals) from a system. Meanwhile in some literature, the machine learning methods refer to a narrow band of models that take numerical input and produce a regression or classification output. These models are often called traditional machine learning models, to distinguish from the later developed deep learning models. Both definitions are used in this dissertation and the meaning is indicated when it is referred to.

The general framework of self-learning diagnosis systems is shown in Figure 1.2.1. A training stage is required for the system to discover and learn feature patterns from the data, while the diagnosis stage is carried out after a model is trained. The processing workflow is shown from left to right. The log data in the time-ordered message sequence format first go through the feature discovery process to obtain distinctive patterns that belong to a fault. Then a learning model fits on the feature patterns in order to produce the fault classification. The same feature discovery process and the trained model are used to process a new piece of log data and produce a diagnosis result.

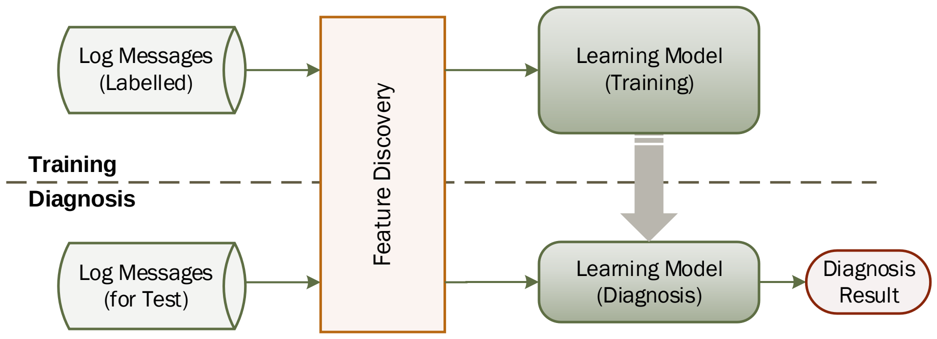


Figure 1.2.1 The general framework of software fault diagnosis systems.

Three different approaches are investigated under this self-learning framework, namely:

1. data mining;
2. statistical machine learning; and,
3. deep learning.

They range from intuitive to abstract, representing the progress of this research from a simple to a more complex level. Different feature discovery methods and learning models are used for each approach. They are briefly introduced as follows:

1. The first proposed method is named Fault Diagnosis via Sequence Pattern Mining (FDSPM). The concept is intuitive and straightforward: the feature discovery mimics test engineers' manual process of examining the log data by using data mining methods. Data mining refers to the algorithms and methods of finding patterns from a large dataset. In particular, the sequential data mining algorithm is selected for the pattern discovery process. The probabilistic weightings of the discovered patterns are determined by observing the training data. In the diagnosis

phase, a Bayesian inference model utilizes the pattern weightings and produces the likely fault with probability rating.

2. The second method is called Fault Diagnosis via Statistical Machine Learning (FDSML). The concept is to use statistical representations to describe log sequences as vectors. This turns the data into a numerical form and renders many traditional machine learning classifiers applicable. It is the most practical approach in big data applications and possibly the most popular one in industry. In particular, a Multi-Layer Perceptron (MLP) model is investigated and evaluated on the Ford log data.
3. Deep learning is the third approach presented in this research. Deep learning refers to the recent advancement of neural network development, which can be viewed as a complex version of machine learning models. The concept is also to convert categorical data as numerical, but with deep learning, the feature representation goes one step further by converting log sequences into 2D matrix features, a technique called embedding. Specialized deep learning models handle these matrix features and perform effective classification.

Deep learning models can be highly flexible and capable of including various information, such as timestamps. This study also investigates the effect of time information on the analysis of log data and extends the existing deep learning models to incorporate such information.

1.3 Contributions and Novelty

The contributions of this research are as follows.

This study is the first work to examine Ford SYNC system log data for fault diagnosis purposes. The work provides guidelines on processing log data using AI methodologies to reduce repetitive workload during software development and testing.

The second contribution is the novel application of sequential data mining to system log data. The proposed FDSPM method combines an effective mining algorithm and the Bayesian classifier to diagnose reoccurring faults. This intuitive method has demonstrated effectiveness even with a limited amount of data.

The third contribution is the application of machine learning and deep learning approaches. These approaches use well-established methods, including log vectorization, logistic regression, word embedding, and neural networks. For the first time, this study brings these efficient methods together and applies them to the Ford SYNC log data.

When investigating the log data, a novel finding reveals that the log sequence durations have distinctive distribution patterns between normal and faulty data, implying their usefulness in fault detection. Inspired by this discovery, the fourth contribution is a novel neural network structure called Ts model that incorporates the timestamp information of log sequences. The Ts models demonstrate improvement over the regular deep models, confirming the contribution of timestamps and the effectiveness of the proposed models.

1.4 Thesis Organization

This dissertation is outlined as follows.

Chapter 2 provides an overview of Ford's defect management process and presents the fault identification and diagnosis problem in more details. The log data and the analysis workflow are also introduced. Chapter 3 gives a review of the literature related to system log data and software defect diagnosis. Chapter 4 presents the Fault Diagnosis via Sequence Pattern Mining (FDSPM) method and the data mining approach applied to the fault identification problem. Various data mining approaches are reviewed and one algorithm is selected for extracting the feature patterns for Ford SYNC log data. The Bayesian classification is used to extracted patterns. Chapter 5 presents the traditional machine learning approach using statistical features and proposes the Fault Diagnosis via Statistical Machine Learning (FDSML) method. Statistical features that are commonly seen in recent machine learning-based log analysis research are introduced. The FDSML is implemented and evaluated against the Ford SYNC log data and an augmented version of the dataset. Chapter 6 discusses a proposed deep learning approach and its application to system log data. The process of "embedding" that is a comprehensive vectorization method is introduced and implemented for system log data. Two deep learning models are then implemented to process the embedding vectors. Chapter 7 presents a novel strategy using time information in relation to log data. The significant differences brought by log timestamps are presented both statistically and descriptively. The novel deep learning structure that incorporates timestamps is proposed. The results are compared with the regular deep learning models reported in Chapter 6. Chapter 8 presents the conclusions and future research directions.

Chapter 2 System Log Data Overview

In modern computer software, system logs are generated constantly by various operating systems. These logs contain event traces and software running status, providing important information for developers to troubleshoot issues. The task of log data analysis requires domain expertise and therefore, is traditionally done by experienced software developers. However, it becomes increasingly more challenging as the systems grow in scale and complexity. The modern vehicle infotainment system, or in-vehicle entertainment system as an example, is one of such operating systems. It performs functions that include audio processing (radio/CD/auxiliary), navigation, USB and Bluetooth connectivity, Wi-Fi, and voice control. These systems have evolved over time with new features being continuously added and updated as consumers' requirements evolve.

This chapter takes Ford's infotainment system as an example to provide an overview of what system log is, the role it plays in the software development process, and the benefits of automating log data analysis. The problem of defect diagnosis is stated in the end.

2.1 Definitions

The terms *defect*, *fault*, *error,* and *failure* appear commonly in the literature, but their definitions depend on the type of the system as well as conventions. One general definition by Naik and Tripathy is as follows [7]:

- A *failure* is used when the behaviour of a system does not conform to what is specified in the requirement or what the customer expects.
- An *error* is a state of the system that could lead to a failure if not corrected.
- A *fault* is the adjudged cause of an error.

The definition of the term defect varies depending on the system. In mechanical or electronic systems, a defect often means a flaw or imperfection of a physical component. Typically, the causal chain is defect → fault → error → failure. Take the internal combustion engine system as an example, a *fault* can be the physical degradation of spark plugs that do not meet technical specifications. It may cause uneven combustion which is an *error* state. During this error state the system would have degraded performance which may be noticed by the driver or discovered by a technician. If no actions are taken, the engine would eventually *fail* due to no ignition or excessive vibration. The defect in this example, is a *defective* product (spark plug). However, a defect is not a must; a component can be free of defect and the fault caused by normal wear and tear. defects and faults often have clear definitions for such physical systems [8].

In software systems, a *fault* commonly refers to a program code that does not meet specifications. When the faulty code is executed, it brings the program to an *error* state.

The error may or may not show up immediately – the latter is called latent error. The error state eventually presents in the form of a *failure* by customers or by testers. For example, a faulty block of code leads to a user interface widget in an error state. When interacting with a user, the widget causes the software application to crash (fail). Unlike physical systems, many software failures are not as evident and may go unnoticed for a long time. The term *defect* in software systems does not have a clear definition equivalent to a physical system. In fact, the IT industry often uses the term defect equivalently as fault. In this chapter, the two terms are used interchangeably as required.

Figure 2.1.1 [9] shows the relationship between faults, errors, failures and logs in a software system. The chain of faults, errors, and failures does not always happen, so the failures are a subset of errors and errors are a subset of faults. When the failure propagation happens, the log data may directly report it, such as the crash logs always capturing the crash events. In this case the failure is evident and the fault can be traced; otherwise, the diagnosis is not feasible because of a lack of evidence.

A comprehensive diagnostics tool aims to capture as much valuable information about a failure as possible, effectively increasing the size of "event log" oval in Figure 2.1.1. As a result, the event log not only contains errors and failures, it also records a large amount of regular executions. The richer the information in the log data record, the more likely they can capture the propagation of a failure. The unreported failures are not considered in this study.

When a fault exists and a failure occurs, the troubleshooting process often includes *detection* and *diagnosis*. Detection is the process of observing and uncovering the abnormal behaviour of a system. Diagnosis is to determine the root cause – the actual fault. The term Fault Detection and Diagnosis (FDD) is commonly used in physical systems, while software systems use a different convention. In software systems, *failure detection* is an important task as the failures can happen without being noticed, so quick detection and resolution are of great benefit to system maintenance. On the other hand, software *fault diagnosis* refers to finding the root cause in the source code. It is also called fault localization. A general approach is to trace the code execution using the evidence provided by log data. This is possible because log messages are often directly linked to the source code.

This study investigates fault detection and diagnosis. Since the source code is unavailable due to corporate confidentiality, a thorough fault diagnosis that pinpoints the fault codes is not feasible. Instead, the focus is to identify reoccurring faults that 1) are previously known and manually diagnosed and, 2) leave traces in the log data. The term fault diagnosis or defect diagnosis is used throughout this thesis. However, it should be noted that when using this term, it only applies to the eligible faults that satisfy the above two conditions.

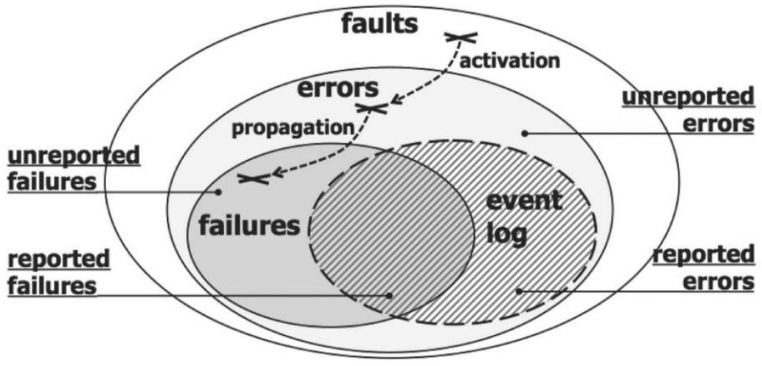


Figure 2.1.1. The fault-error-failure chain in software systems [9].

2.2 Ford System Log Data and Software Defect Diagnosis

In Ford Motor Company, the infotainment system, i.e. the Ford SYNC® 3 – and the SYNC® 4 by the time that this dissertation is finished – is based on the QNX system, a Unix-like real-time operating system. To assist software development, a data analytics team has developed and maintains a logging framework for defect diagnosis and management purposes. The framework provides data logging functionality during the development phase. After product release, the same framework may also provide analytics data and serve customer support to help troubleshoot user complaints. The current process of defect diagnosis is elaborated in the following subsections.

2.2.1 User Acceptance Test

In the software development life cycle, the user acceptance test is the stage that follows after all subsystems are integrated and before product release. This stage is mainly concerned with system integration and is to make sure that different software and hardware modules function properly when working together. System-level validation tests and real-world use cases are performed to identify possible issues and bugs before the product is released to the market. This stage includes alpha testing and beta testing, referring to the software versions being tested: alpha and beta. The alpha testing is performed by testers within the organization. After its completion, the beta testing is carried out by selected end-users outside of the organization. The overview in this chapter mainly concerns alpha testing, though many facts also apply to beta testing.

Alpha testing of the SYNC system is mainly achieved through two means: the bench test and the road test. Either way, the test is performed by a tester who is a trained testing engineer. In the road test, the tester will drive the vehicle and use the infotainment system in a similar way as a regular consumer, such as driving on highways while using hands-free functions to make a phone call. The software under development carries a bug report module that constantly writes the log data in a buffer. When a suspected failure is observed – this could be as simple as something not feeling right – the tester would flag it through the SYNC's touchscreen interface. The log data in the buffer are saved, along with a snapshot of the system's current status, including a screenshot, date and time, and the vehicle ID. This set of records is called a *bug report*. It is automatically uploaded to the organization's server for further processing.

2.2.2 Bug Report Triaging

The automatically created bug report usually captures the fault to failure propagation – details elaborated in Section 2.2.3 – but it does not contain explicit information about the encountered failure. It also could contain false reports that the tester mistakenly triggers. For example, a tester may misunderstand the requirement and reports a fault. In other words, the bug reports are not complete. Therefore, annotations are often manually added to assist post-analysis. The tester would describe the use case and the experienced anomaly in the form of voice recording or text and append them to the bug report. This annotation process also helps to remove the reports that are falsely triggered.

The bug report is then assigned to the *triage* team through a *defect management system*. The task of triaging is to pre-examine and categorize the bug reports. Instead of sending bug reports directly to the software developers for troubleshooting, this intermediate triaging process is necessary for several reasons. First, multiple function teams work in parallel to maintain one large system like the SYNC, each focusing on its own software module, which is also known as a software component or subsystem. A bug report needs to be categorized before being sent to the right function team. Secondly, multiple bug reports can relate to the same fault. This is caused by a tester reproducing the same anomaly and creating several reports, or the same issue reported by more than one tester. Grouping these bug reports as one defect helps reduce redundancy. Lastly, the triaging process determines the defect severity level and filters out false alarms, which optimizes the allocation of the software development work.

2.2.3 Software Defect

A software *defect* as explained in Section 2.1, is the root cause of unexpected software behaviours. In the Ford case, an unknown defect causes a failure that is perceived by the tester, and a bug report is created to initiate the process of defect diagnosis.

After the triaging process of sorting and allocating bug reports, the team has a better understanding of what the defect is and how it behaves. This defect is called *identified*.

An identified defect is documented with a defect ID along with many details, such as the description, ways to reproduce, linked bug reports, severity, and priority. A well-documented defect is then assigned to a *software developer* in the corresponding function team. The developer's responsibility is to find the root cause and implement code changes to fix the defect.

A defect record is a live document as shown in Figure 2.2.1. New bug reports could be added to an existing defect if they have the same root cause. The developer may in turn request test engineers to produce more bug cases to confirm an error is reproducible. A defect management system is utilized to track and report the status of defects and bug reports. Figure 2.2.2 illustrates the forward workflow of the bug reporting process.

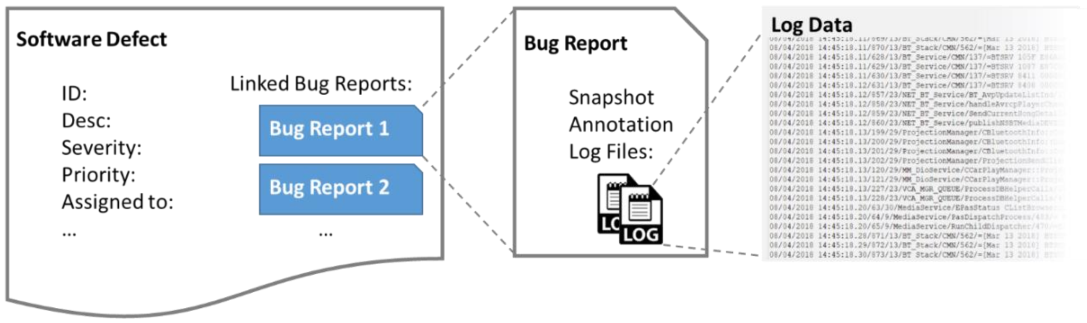


Figure 2.2.1 The anatomy of a software defect in the defect management system.

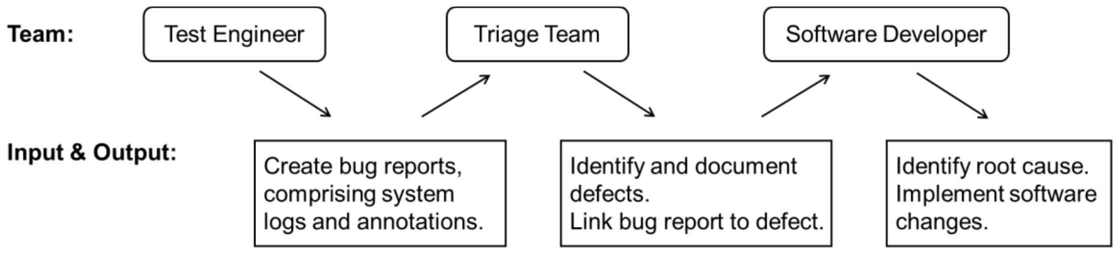

Team:
Test Engineer
Triage Team
Software Developer
Input & Output:
Create bug reports, comprising system logs and annotations.
Identify and document defects. Link bug report to defect.
Identify root cause. Implement software changes.

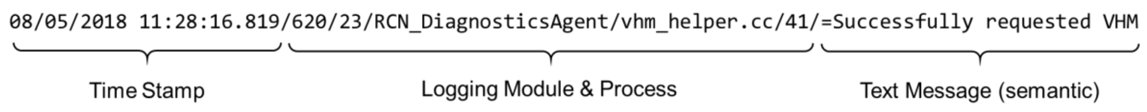
Figure 2.2.2 The workflow of software defect management.

2.3 Log Data Overview

Among the masses of relatable information within a bug report, the system log data is the "heart and soul" – quoting from the software engineers – for the task of triaging. This is because the descriptions and annotations from the testers only scratch the surface. The underlining erroneous module often needs to be traced down through the log data.

The log data can appear in vastly different formats depending on the logging purpose and how the logging framework is designed. The SYNC system maintains several log buffers for various purposes, such as the network status, the memory usage, and the status of external devices. Each of them has a unique logging format, some are hardly readable for non-experts in the domain. Fortunately for the triaging task, the most used – and probably the only useful one – is the *software debug log*, which records the operations of each function module. It comprises a sequence of readable messages in the order of timestamps. In the following discussion, the wording "log file" or "log data" refers to the software debug log, unless otherwise noted.

The software debug log generated by the SYNC system is the typical Unix style system log. The log data consist of individual *log messages*. When the system is running, log messages are constantly buffered in the background. Each message contains a timestamp, logging module and process, and a text message. Figure 2.3.1 shows a typical log message that contains semantic texts.



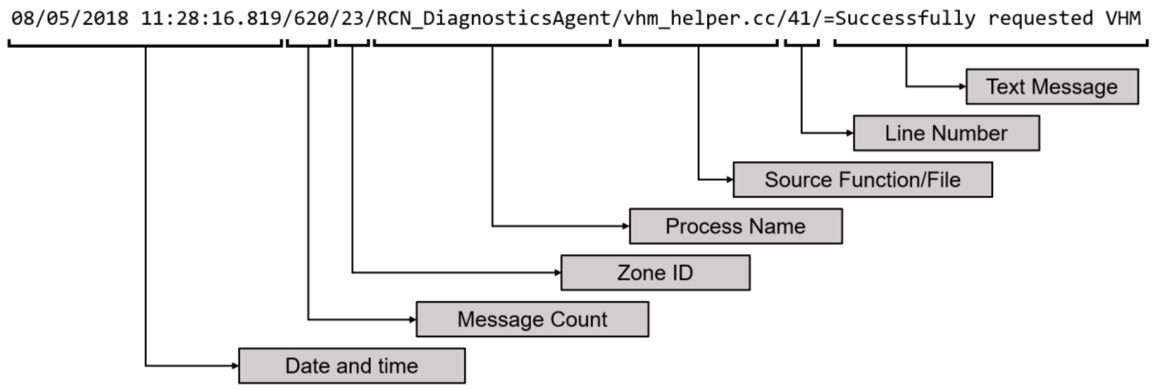
```
08/05/2018 11:28:16.819/620/23/RCN_DiagnosticsAgent/vhm_helper.cc/41/=Successfully requested VHM
```

Figure 2.3.1 An example log message from the SYNC system.

The buffer has a fixed size (measured in Megabytes), containing a fixed number (thousands) of messages. When the system user triggers a bug report, the log buffer's content is saved as a text file. Depending on the software workload at the time, the file can record up to an hour of runtime. This duration is usually sufficient to capture the fault to failure propagation chain.

The log message itself contains rich information for debugging purpose. It shows which software module produces the message, pinpointing the line of code of the source file. A detailed breakdown of a log message is shown in Figure 2.3.2. These fields include:

- Timestamps containing both date and time, accurate to milliseconds.
- Message Counter that is an incremental counter, which resets after reaching a max limit.
- Zone ID that is a hard-coded number for each task.
- Process name that represents the software feature. It often corresponds to software function teams.
- Source function/file and line number location in the source code that prints this message.
- Text message that is the output of the log printing command. It is often semantic, while in some rare cases encrypted.



```
08/05/2018 11:28:16.819/620/23/RCN_DiagnosticsAgent/vhm_helper.cc/41/=Successfully requested VHM
```

Figure 2.3.2 The structure of one log message.

2.4 Problem Description

Defect diagnosis, or identifying the defect during the triaging process, is one of the challenges in the defect management workflow. The engineers need to closely examine the bug reports – mostly including reading the log messages line-by-line – in order to identify the defect. Deciphering the sequence of log messages not only requires domain knowledge, but also finding evidence from tens of thousands of log lines can also be time-consuming and tedious.

The challenge also comes from the growing size of the software as more and more features are added. This requires extensive tests to ensure the system meets customer satisfaction, likely leading to more bug reports to be analyzed. Additionally, as the acceptance test rolls into beta stage, the clients instead of testers will take the lead and carry out the tests. The amount of bug reports increases significantly and many of them are likely associated with the same defect, meaning heavier and more repetitive work for the triage team.

Some software tools are adopted to improve the efficiency of the manual process of identifying a potential defect, such as using a text viewer with convenient searching functions. Keyword searching can be effective to locate some errors if they are directly printed out in the logs. However, in most cases, the indication of a defect is implicit. More commonly a sequence of special log messages, often referred to as the log pattern, is the key to identifying a defect. The individual messages may appear benign, but an experienced triage engineer is able to identify the software actions behind the patterns and correctly diagnose a defect. In these cases, simple keyword matching does not function properly.

An automated defect diagnosis tool would be beneficial to the triaging process. This tool is expected to analyze the bug report – specifically the system log data – and detect and identify potential defects within the system. If such a program is put into place, the triage engineer would see suggestions of possible defects before examining a bug report. This would be much relief as the engineer would only need to verify the suggested defect, instead of looking aimlessly into the bug report and log data.

The following assumptions are made regarding the defect and system log data:

- when a defect is triggered, it leaves traces in the log record, i.e. the defect is observable through the log data; and
- the defect’s trace is captured within the recording buffer period.

Defects that do not conform to these assumptions are excluded. They may include cosmetic defects of the Graphical User Interface (GUI), such as a misalignment of a touch button, or the defects taking too long to reveal, such as bad memory management causing the out-of-memory error after the system is used extensively.

In this dissertation, a few approaches for this software defect diagnosis problem are considered, ranging from intuitive methods, that mimic the engineer, to machine learning methods using highly abstract feature patterns. The next chapter presents a literature review on troubleshooting using system log data.

Chapter 3 Log Data Analysis Literature

This chapter reviews various system log data and common analysis approaches in automated failure detection and fault diagnosis studies. Log data are essentially unstructured or semi-structured text data in a sequential format. The appearance of system logs evolves over time as newly developed systems bring more complicated logging functionality and a significantly larger amount of log data. Methodologies developed for an earlier system may not perform well on the later systems and their log data; one reason being the recent systems often log extra information that requires special handling. Therefore, the types of log data need to be introduced before discussing methodologies. An overview of various types of log data commonly seen in industry is presented in Section 3.1.

Automated failure detection and fault diagnosis through data analysis are typical classification problems that are commonly approached with machine learning [10]. The detection and diagnosis can be viewed as a supervised task, where a learning model trained on labelled data produces classification results. Alternatively, the problem can be handled in an unsupervised way, where the model analyzes the patterns of unlabelled data and detects failure conditions as outliers [11]. The classification strategies currently being used show a clear trend of development: earlier works tend to focus on the sequential order of log messages and are referred to as *path-based* methods; the more recent methods of machine learning are based on *statistical features* that are extracted from log samples; the emerging *deep learning* models directly handle log sequences, capable of incorporating variables within the log messages as well. These works are reviewed in Sections 3.3 through 3.5, respectively.

Note that existing research regarding log data analysis also involves other topics, such as logging mechanism [12], [13], infrastructures [14], security [15], and log filtering [16]. These topics are beyond the scope of this review.

3.1 Types of System Log Data and Their Applications

According to the survey paper in [17], the research published on work related to the topic of log data analysis shows a growing trend over the past two decades as shown in Figure 3.1.1. According to this survey, the first publication on log data analysis dates back to 1997, while a more rapid uptake of research in this area starts around 2014. Note that this survey was published in mid 2020, so the publication counts for the year 2020 is likely incomplete. The log data research has an upward trend to date and is likely to keep growing.

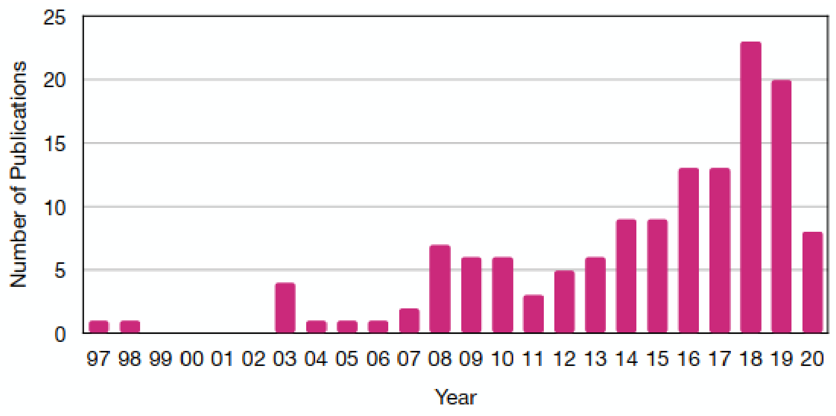


Figure 3.1.1. The evolution trend from 1997 to 2020. (Year 2020 is likely incomplete.)

Before the term *system log* was widely used, studies investigated the *alarm messages* in large-scale communication networks with the goal of identifying system faults back in the 90s [18], [19]. In the example from IBM in [20], the alarm message describes the anomaly observed by a submodule in a network. These alarms provide information about *who* (the subsystem affected), *what* (the symptom of the fault), and *when* (time occurred). The fault identification task is to determine the *where* (the location of the fault) and *why* (the nature of the fault) factors. Diagnosing a fault often requires analyzing a series of alarms where the sequence information can be important. These alarm messages can be viewed as primitive log data as they are the trace of the system. The difference is that the alarm messages are often well-defined by communication protocols [21], [22], while log data are less organized and prone to change. Figure 3.1.2 shows the transition of two system states (indicated in the brackets) producing certain alarm messages (indicated above the arrow) defined by the IEEE 802.2 logic link control protocol [22].

$$(\mathbf{ERROR,ERROR}) \xrightarrow{\text{R\_DISC\_CMD/S\_UA\_RSP}} (\text{ADM.SETUP})$$

$$(\mathbf{ADM,SETUP}) \xrightarrow{\text{R\_SABM/S\_UA\_RSP}} (\mathbf{NORMAL,SETUP})$$

$$(\mathbf{NORMAL,SETUP}) \xrightarrow{\text{R\_SABM/S\_UA\_RSP}} (\mathbf{NORMAL,SETUP})$$

$$\cdots$$

Figure 3.1.2. Examples of state transitions and alarm messages [22].

With the prevalence of the World Wide Web, log data saw its biggest application on Internet service systems, where the term *log* was widely used. Examples include e-commerce websites [23], peer-to-peer systems [24], and voice service applications [25]. The associated data are often called *request logs*, where the term *request* means the one-way or two-way communication between a user and a subsystem, or two subsystems. The request logs are echoes of the software variables, so the log message is in a structured or semi-structured format. A sample log message from an e-commerce web application [26] is shown in Figure 3.1.3. The two variables – `Machine` and `RequestType` – and their values are well-defined and easy to understand.

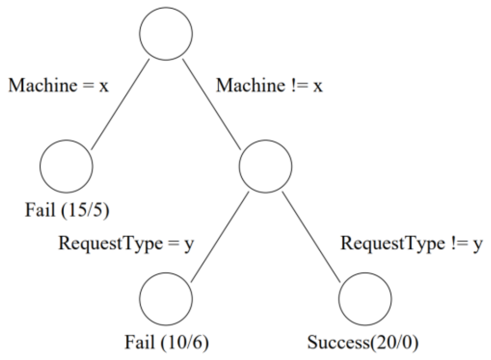


Figure 3.1.3. Analyzing the request log using a tree-based method [26]

The more recent and complicated log data are from various large-scale systems, including supercomputers [27] and cloud computing infrastructure [28]. The form of log data becomes less structured – many of them being completely unstructured free text and only meant for human developers to read – posing increasing difficulty for automated analysis. This form of log data needs to be converted into a structured form, such as extracting the log message template and log variables. This process is called log parsing. This dissertation reviews parsing research in later chapters where related.

An example of a cloud computing system is the Hadoop Distributed File System (HDFS) log that is shown in Figure 3.1.4 [29]. This log message describes an action related to a *storage block* (identified by the ID `blk_801792886545481534`) and an *IP address*. These two parameters are easy to extract for this example, but log messages describing different actions often have different sentence structures. Precisely extracting the useful information is a critical step to analyzing such logs.

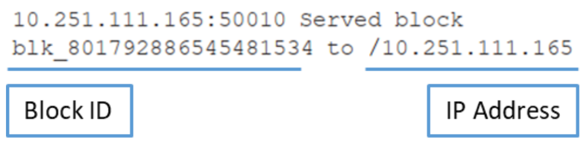
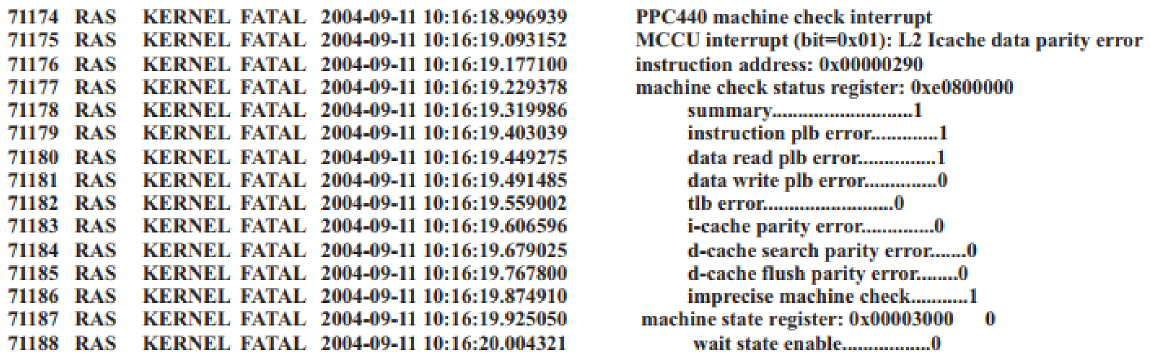

```
10.251.111.165:50010 Served block
blk_801792886545481534 to /10.251.111.165
```

Figure 3.1.4. A typical HDFS system log message [29].

The log data from many operating systems are semi-structured, such as the event logs from the BlueGene/L supercomputer [30] shown in Figure 3.1.5. In this snapshot, the first half of each entry is structured, including id, type, facility, severity, and timestamp. The rest of the log message is unstructured: note that each line has a different sentence template.



```
71174  RAS   KERNEL  FATAL  2004-09-11 10:16:18.996939     PPC440 machine check interrupt
71175  RAS   KERNEL  FATAL  2004-09-11 10:16:19.093152     MCCU interrupt (bit=0x01): L2 Icache data parity error
71176  RAS   KERNEL  FATAL  2004-09-11 10:16:19.177100     instruction address: 0x00000290
71177  RAS   KERNEL  FATAL  2004-09-11 10:16:19.229378     machine check status register: 0xe0800000
71178  RAS   KERNEL  FATAL  2004-09-11 10:16:19.319986         summary..........................1
71179  RAS   KERNEL  FATAL  2004-09-11 10:16:19.403039         instruction plb error............1
71180  RAS   KERNEL  FATAL  2004-09-11 10:16:19.449275         data read plb error..............1
71181  RAS   KERNEL  FATAL  2004-09-11 10:16:19.491485         data write plb error.............0
71182  RAS   KERNEL  FATAL  2004-09-11 10:16:19.559002         tlb error........................0
71183  RAS   KERNEL  FATAL  2004-09-11 10:16:19.606596         i-cache parity error.............0
71184  RAS   KERNEL  FATAL  2004-09-11 10:16:19.679025         d-cache search parity error.......0
71185  RAS   KERNEL  FATAL  2004-09-11 10:16:19.767800         d-cache flush parity error........0
71186  RAS   KERNEL  FATAL  2004-09-11 10:16:19.874910         imprecise machine check...........1
71187  RAS   KERNEL  FATAL  2004-09-11 10:16:19.925050      machine state register: 0x00003000      0
71188  RAS   KERNEL  FATAL  2004-09-11 10:16:20.004321          wait state enable................0
```

Figure 3.1.5. A snapshot of the BlueGene/L supercomputer performance log [30].

Despite different applications, the logs from operating systems and cloud infrastructures share a lot of similarities in terms of the format. In fact, many modern software systems have a similar log appearance [27], [31], including the Ford SYNC system log shown previously in Figure 2.3.1. This indicates that a method developed for one application is likely to be applicable to other systems, though the performance may vary.

3.2 Rule-Based Analysis

Detecting a system failure based on a set of manually specified rules is called the rule-based approach. Although not the intention of this review, the rule-based approach needs to be mentioned as it is the most intuitive and widely used method in industry. It often comes in the form of an expert system, where a human expert establishes a set of rules using regular expressions [32]. The ruleset is used to match incoming log messages to detect failures or other specified events. Once the ruleset is defined, detection or diagnosis can be automated with various data processing tools, such as Swatch [33], SEC [34], and Logsurfer [35]. Its drawback is also evident: the expert system requires immense effort to make and maintain a comprehensive ruleset because of the fast iteration of software and the rapid growth of log data. Academic researchers have generally steered away from this approach.

3.3 Path-Based Analysis

The path-based approach investigates the explicit software execution path to determine a failure in the system. This approach is also intuitive, as unsuccessful executions of a program often produce log messages that reveal deviations from the regular execution path. Each log message is treated as an identifier – in some cases, the logs are already uniquely identified [22] – and a workflow model is created by observing log sequences. Based on how the data is utilized, these studies fall into two categories: modelling regular workflows and modelling failure workflows.

3.3.1 Modelling Regular Workflows

Figure 3.3.1 shows a regular program workflow with a loop [36], where nodes s_0 through s_4 represent system states and links A – D represent the transitions recorded by log data. Specifically, when the system transitions from state s_0 to s_1, it produces a trace that is recorded by log message A. The notation ε indicates that the transition leaves no trace in the logs. The detection is performed by checking if a new log sequence obeys this workflow. Without the source code, the system states are unavailable or hard to obtain [37], so the workflow model needs to be constructed by observing sequences of logs, i.e., the conditions A – D in Figure 3.3.1.

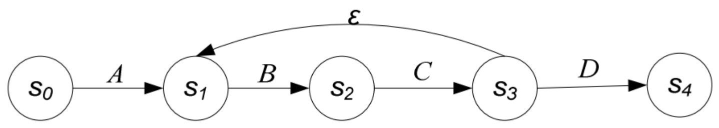


Figure 3.3.1. Workflow model constructed from log sequences [36]

The Finite State Machine (FSM), or Finite State Automata (FSA) is a common candidate to model such system workflows. Researchers in the 1980s [38]–[40] studied the modelling of computer networks using FSMs, where the nodes were system states and links were the messages defined by a communication protocol. Fault detection was also investigated. Bouloutas et al. [22] applied an FSM model to detect faults in communication networks. However, these early works assumed that the FSMs were either pre-defined or could be obtained from some well-defined sources, such as standards and requirements documents.

In many modern software systems, the FSM for the program workflow is often unavailable and needs to be synthesized from log sequences. One popular method to infer an FSM from input-output data is the *kTail* algorithm originally presented in the 1970s [41]. The kTail and its variations use a tree representation of sequences and iteratively merge new branches to reach a final automaton. It is still a popular choice for generating workflow FSMs in the log literature. The challenge presented by log data is the interleaving sequences, a result of multi-threaded tasks generating log messages at the same time [36]. Some publications refer to this interleaving log data as heterogeneous logs [42]. Various adaptations of the kTail are developed to remove the false dependencies caused by interleaving sequences [43], [44]. New algorithms are also proposed, such as constructing an FSM using n-grams [45] (n-gram refers to using a sequence of n elements as one unit, introduced in Section 3.4) and refining FSMs with an emphasis on temporal dependencies [36].

The log data used to generate FSMs are preferred to be regular log data generated by error-free task executions. However, the log data inevitably contain incomplete workflows and traces of fault propagations. Using these noisy data to generate a deterministic FSM could lead to an inaccurate representation. Probabilistic models are investigated to address this issue, such as probabilistic FSMs [46]. One special case of the probabilistic FSM is the Hidden Markov Model (HMM), where the future state only depends on the current state. In particular, Yamanishi and Maruyama [47] constructed multiple HMMs and proposed a dynamic model selection method to pick the best result. Lim et al. [48] proposed a clustering model for known faults using the Hidden Markov Random Field (HMRF) approach.

Without an FSM, the path-based methods are few. Invariant mining is a kind not relying on an FSM [49], [50]. It implicitly models the system execution paths in the form of linear relationships (invariants) of the logs. Take the workflow in Figure 3.3.2 as an example; the letters A to G represent the program states, which generate corresponding log messages. If the paths represented in the figure are complete, the count of each log message will meet certain conditions. For example, the counts of A, B, and G should be the same, and the count of B should equal the sum of C, E, and F. Obtaining this set of linear relationships effectively models program execution workflows in a numerical form. An anomaly is detected if a new log sequence does not comply with this set of linear rules.

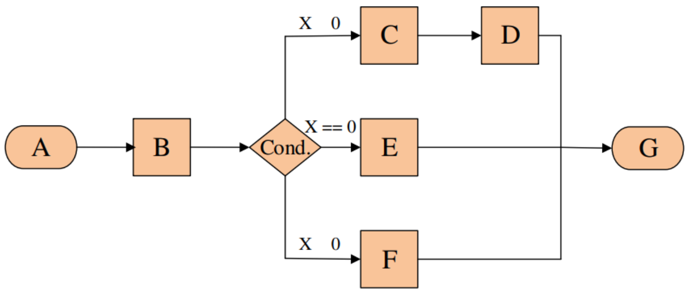


Figure 3.3.2. A program workflow example with if-conditions [51]

3.3.2 Modelling Failure Workflows

The path-based approach does not necessarily model the entire program in order to perform automated failure detection. The opposite way is to only learn the faults' behaviours by collecting and observing faulty log sequences. One example is to use the decision trees [26], where the branches represent the failure execution conditions. This simple technique is proven to be effective to detect failures in an online system [51].

Additionally, extracting partial workflow paths that represent some characteristics of a system can also be effective in fault diagnosis. In other words, the segments of a workflow path can be a distinguishing representation of the workflow. In particular, Yu et al. [52] extracted distinctive sequence segments and used a graph-based approach to represent various execution paths.

3.3.3 Discussion

There are several differences between modelling regular workflows and failure workflows. First, regular workflow models are mainly used for detection, whereas failure workflows provide classification or diagnosis capability. Secondly, the classification tasks typically require labelled failure log data, hence it is a supervised learning task.

Modelling regular workflows is largely unsupervised, only requiring regular log data. Additionally, working directly with failures generally shows higher accuracy compared to the regular workflow approaches. In practice, however, obtaining the failure samples can be difficult as faults are often rare and unpredictable. Lastly, the use of failure workflows is only limited to detecting and classifying known issues. Unseen failures cannot be learnt without sufficient prior knowledge. This is not an issue for regular workflow models, which can theoretically detect all failures as outliers.

It is worth noting that many path-based approaches introduced in this section are applied to relatively simple log datasets, as one shown previously in Figure 3.1.3. Recent large log datasets are few, potentially because scaling up these models causes complexity or efficiency issues.

3.4 Statistical Feature Extraction and Machine Learning Approach

With the drastic increase in the quantity and complexity of log data files in recent software systems, such as in cloud computing and operating systems, the path-based approach can become less efficient or even infeasible as logs from different workflows are interleaved [36]. The relatively new analysis approaches use statistical features instead of sequential features. Figure 3.4.1 shows a general representation of this approach’s workflow [17]. It typically includes four steps: 1) log partition to divide consecutive log data into samples for training; 2) feature extraction to turn each sample into a numerical representation; 3) a machine learning model to train on numerical features; and, 4) detection as performed by the trained model to process new log sequences.

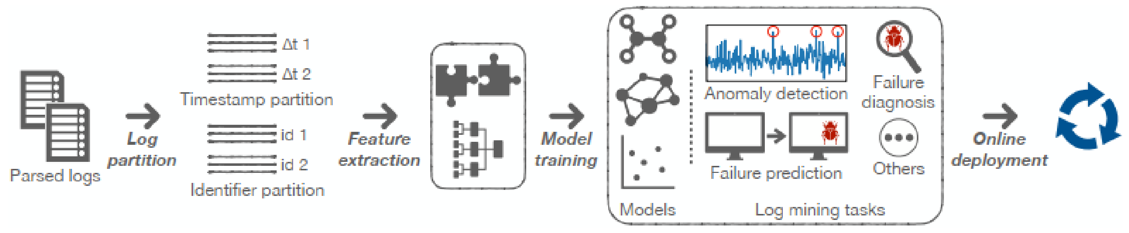
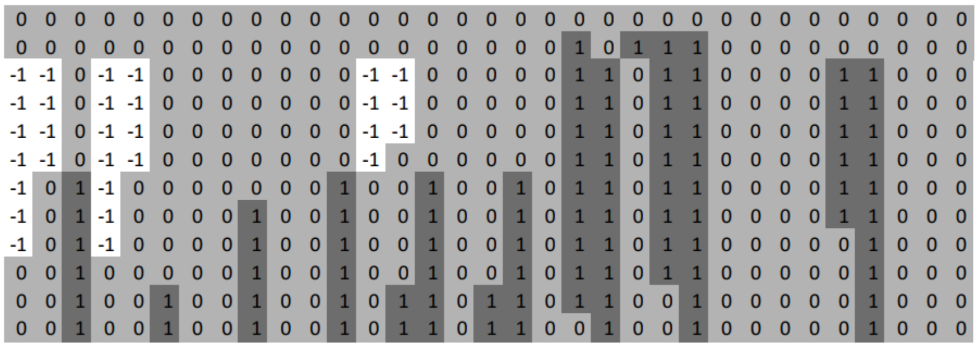


Figure 3.4.1. A general workflow of the feature extraction and machine learning approach[17]

The key process in the pipeline is the feature extraction step. Specifically, a set of quantitative metrics are generated to describe the sequential log data. The feature sets are typically much smaller than the original log sequence. Thanks to the numerical form of the extracted features, many well-known machine learning algorithms are good candidates, such as logistic regression and *k-means* clustering. Both feature extraction and machine learning model training are generally very efficient, so this approach is often used for processing of data that are too massive to manually inspect. Some may call it log mining for this reason [53], but the methodologies have generally diverted away from the traditional data mining area. Existing studies mainly emphasize the feature extraction process and the choice of machine learning models.

Feature extraction is the crucial step of these applications, as the statistical indicators must be carefully picked to appropriately represent the log data, otherwise, important information can be lost during the process. On the other hand, the choice of machine learning models can be arbitrary and less constrained by the type of features. The following paragraphs present a few examples in this research domain.

In the special case where the log data are structured and in a numerical form, such as the performance indicators log in [54], the feature design can be relatively straightforward. Bodik et al. [54] defined a log feature vector called *fingerprint*, condensed from the performance log of a datacentre. The performance log in this application is the numerical metrics that monitor the system’s health. Statistical metrics, such as medians and other quantiles, are used to describe a chunk of log data. The processed fingerprint feature is a 2D matrix shown in Figure 3.4.2. Similarly, in an application analyzing anomalies of a large-scale data analytics engine [55], the feature set contains readily available or easily extracted values, such as memory usage, task duration, and total elapsed time.



0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	1	0	0	0	0	0	0	0	0	0
-1	-1	0	-1	-1	0	0	0	0	0	0	0	-1	-1	0	0	0	0	0	1	1	0	1	1	0	0	0	0	1	1	0	0	0
-1	-1	0	-1	-1	0	0	0	0	0	0	0	-1	-1	0	0	0	0	0	1	1	0	1	1	0	0	0	0	1	1	0	0	0
-1	-1	0	-1	-1	0	0	0	0	0	0	0	-1	-1	0	0	0	0	0	1	1	0	1	1	0	0	0	0	1	1	0	0	0
-1	-1	0	-1	-1	0	0	0	0	0	0	0	-1	0	0	0	0	0	0	1	1	0	1	1	0	0	0	0	1	1	0	0	0
-1	0	1	-1	0	0	0	0	0	0	0	1	0	0	1	0	0	1	0	1	1	0	1	1	0	0	0	0	1	1	0	0	0
-1	0	1	-1	0	0	0	0	1	0	0	1	0	0	1	0	0	1	0	1	1	0	1	1	0	0	0	0	1	1	0	0	0
-1	0	1	-1	0	0	0	0	1	0	0	1	0	0	1	0	0	1	0	1	1	0	1	1	0	0	0	0	0	1	0	0	0
0	0	1	0	0	0	0	0	1	0	0	1	0	0	1	0	0	1	0	1	1	0	1	1	0	0	0	0	0	1	0	0	0
0	0	1	0	0	1	0	0	1	0	0	1	0	1	1	0	1	1	0	1	1	0	0	1	0	0	0	0	0	1	0	0	0
0	0	1	0	0	1	0	0	1	0	0	1	0	1	1	0	1	1	0	0	1	0	0	1	0	0	0	0	0	1	0	0	0

Figure 3.4.2. The fingerprint feature [54] condensed from log data, where each row is a time window, each column is a particular metric.

In a more common setting, the logs are unstructured similar to the HDFS log shown previously in Figure 3.1.4. Each log message contains the constant part, a.k.a. *log template*, and the variable part, including timestamp and other *log variables*. Xu et al. [56] constructed a series of complicated metrics, such as the count, frequency, and ratios of the log templates and variables, to describe such log sequences. Similarly, Liang et al. [57] used the count of events in a time window as log features. This included new event counts, accumulative counts, the change of event counts, the elapsed time since the last fatal failure, and the log template count. Furthermore, the feature extraction work from Kimura et al. [58] emphasized the periodicity and burstiness of log data.

A more general feature construction technique to is the *n-gram* model [59] and its special case, the *bag-of-words* model. These notions originate from computational linguistics and information retrieval research [60], [61]. In the case of the text document, an n-gram is a sequence of n words from the text. Table 3.1 shows an example of extracting word unigrams (n=1) and bigrams (n=2) from a simple text document. The n-gram *terms* are defined as the fundamental elements to describe the document. Unigram terms are unique individual words and bigram terms are unique phrases consisting of exactly two words as shown in the second and third column of Table 3.1. A model describing the terms' statistical characteristics is called an n-gram model. In the special case of n=1, it is called a bag-of-words model. The frequency characteristics can be as simple as the count of the n-gram, while complex weights are more common and better at describing the document. Specifically, Term Frequency (TF) represents the number of occurrences of an n-gram term, and Document Frequency (DF) represents the number of documents that contain the n-gram term. A popular combined metric, the Term Frequency-Inverse Document Frequency (TF-IDF) weighting, is to assign a higher weight to the term with higher TF and lower DF.

Table 3.1 Examples of n-grams extracted from a piece of text.

Text document	Unigram terms	Bigram terms
"The quick brown fox jumped over the lazy dog."	the, quick, brown, fox, jumped, over, lazy, dog	the quick, quick brown, brown fox, fox jumped, jumped over, over the, the lazy, lazy dog

The n-gram models can apply to letters, words, speech phonemes, as well as log messages. A general approach to model log data is to treat each unique log message as a word in a text document. Then the bag-of-word model and TF-IDF-like weighting methods are applicable and shown to be an effective feature extraction approaches [62], [63].

A machine learning model then processes these features for classification. The machine learning models can be either supervised or unsupervised depending on whether labelled log samples are available. Common unsupervised techniques include Principle Component Analysis (PCA) [56] and similarity clustering [62]. Supervised models can be logistic regression [54] and Support Vector Machine (SVM) [58], [63], [64]. Many of these machine learning models can be used interchangeably, as demonstrated in a comparative study in [51].

The biggest advantage of the feature extraction and machine learning approach is its efficiency in processing a large volume of log data. This meets the growing demand for modern system logs. On the downside, the feature extraction process can only be performed on the sequences with a certain length. As a result, the log data need to be segmented either by time windows or by log sequence lengths, and the detection or classification is performed based on segments. Setting a longer window or larger segmentation tends to produce better accuracy, but increases the response time for system maintenance. Additionally, most feature extraction methods treat the log sequences as a collection of log messages, losing the time and order information that could be valuable for detection and diagnosis.

3.5 Deep Learning Approaches

Neural network models are the new candidates in the log analysis area. Despite their early appearance in the 1940s, neural networks only saw their full potential after the very recent breakthroughs in computational capability [65]. The term deep learning and neural networks are strongly related, as deep learning refers to the recent form of neural networks with an emphasis on a large number of layers and the ability to train such a model. Deep learning models are highly versatile in terms of processing various types of data. Two representative models are the Convolutional Neural Network (CNN) and Recurrent Neural Network (RNN). CNN is the most popular choice in image processing applications [5], [66], while RNN is preferred in natural language processing [67], [68]. Both methods are explained in more details in Chapter 6.

The deep learning models capable of processing sequences are called *sequential* models. They are being investigated in many areas, such as text classification, machine translation, and time series prediction. Their applications to log sequences mainly have two approaches: 1) using the model as a *predictor*, then reporting an anomaly if the actual log message deviates from the model output; 2) using the model as a *classifier*, which directly outputs a failure decision after processing a sequence of logs. Generally, the second approach achieves higher detection accuracy, while the first method is more suitable for online real-time applications.

3.5.1 The Predictor Approach

The predictor approach requires item-by-item processing of the log sequence, so a majority of the studies adopt the RNN model. Figure 3.5.1 shows a typical RNN structure in the predictor setup. The data flow from left to right represents the progression of time, each step $t-1, t, t+1, t+2, \ldots$ is called a timestep. The input log sequence is broken down and processed by the RNN one log message each timestep. The RNN cell highlighted in the middle includes a memory called the hidden state and a series of calculations to update the hidden state. When progressing over time, the cell's hidden state is updated by the input and produces an output at each timestep. The RNN cells are trained in a way that the output vector represents the next value in the sequence, effectively predicting the next item in the sequence. Since the number of outputs equals the number of inputs, this is called a many-to-many structure.

Zhang et al. (2016) were one of the early adopters of the RNN predictor approach. In this work, the log data are segmented by time windows. Each time segment is treated as one unit, and a vector of TF-IDF features is extracted from it. An LSTM model processes the feature vectors in time and generates an output that represents the feature pattern of the next time window. This approach utilizes the similar – if not the same – statistical features introduced in Section 3.4. The sequential order of time window segments is successfully captured. However, the log messages within a time window are represented statistically rather than sequentially.

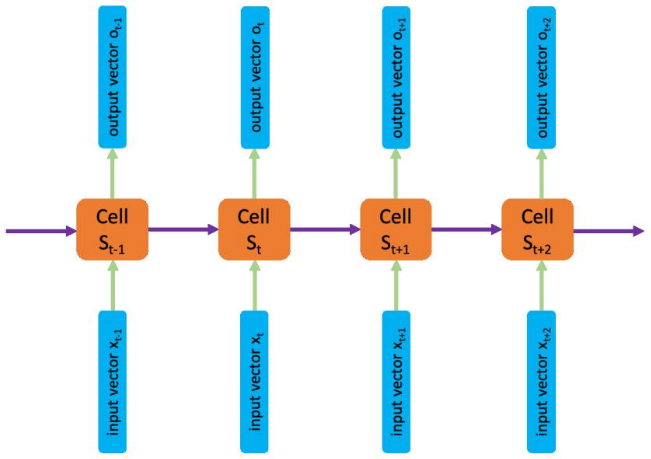


Figure 3.5.1. The predictor style RNN (also called sequence-to-sequence structure) [69]

A more common and effective way to utilize RNN is to treat individual log messages as the basic sequence elements. Du et al. [70] and Frank et al. [71], [72] first converted the log messages into *tokens*, then mapped them into *embeddings* in the numerical form so that they could be processed by an LSTM model. The embeddings, also known as word embeddings or word vectors, are the numerical representation of a set of categorical data in a high dimensional space. The methodology to obtain the embeddings originates from language modelling research [73]. The output of LSTM in these studies is also an embedding that represents possible log tokens. Failure detection is achieved by monitoring the upcoming log message, which would be flagged as an anomaly if it deviates from the LSTM's outputs. Both [70] and [71] further included a separate LSTM model to process timestamps, which were in the form of delta time between log messages. However, the timestamp model is detached from the log-processing LSTM detector, meaning that the timestamps and the log messages are no longer associated. The benefit of timestamps processing is unclear and not evaluated in these works.

Xia et al. [74] adopted the latest generative adversarial network (GAN) – which was originally used to produce image variables – to generate fake log event sequences. These artificially generated log sequences helped to address the lack of abnormal samples. An LSTM model was trained to discriminate the fake sequences from the real ones. Timestamps were used to segment log sequences before input into an LSTM model, but were not directly used by the model.

3.5.2 The Classifier Approach

The classifier models directly output a classification label, such as normal – 0 and failure – 1. Take the example in Figure 3.5.1, instead of producing one output at each timestep, a classifier approach only requires the last hidden state to produce one output, which is trained to match the classification labels. This setting is therefore referred to as the many-to-one RNN structure.

Zhang et al. [75] adopted this approach with a bi-directional LSTM structure, another variation of RNN. The study also developed a novel vectorization method called semantic vectorization, which was based on the individual words in a log message instead of word embeddings used in almost all other studies. This strategy uniquely addresses unseen log messages because unseen messages are constructed by known words. Unseen log messages are rare in the real world, so the study has to artificially inject anomalies to amplify the effect.

One advantage of the classifier approach is that it is not limited to RNN. The CNN sequential model can also be very effective. This variation of CNN, called 1D-CNN, has convolutional kernels that sweep in one direction. Lu et al. [76] proposed a failure detection model that utilizes three 1D-CNN layers in parallel as the backbone structure. He argued that the CNN model was easier to tune than the LSTM equivalent. These classifier approach studies ignored the log timestamps as well.

3.6 Discussion

Three automated approaches – the path-based, statistical feature based, and deep learning – were discussed in this review. Most of them could potentially apply to the defect diagnosis problem described in Chapter 2, with adaptations to various extents. In the meantime, few studies have investigated the log data from operating systems, let alone a vehicle infotainment system. Therefore, investigating the Ford data can be a good supplement to the current literature, in addition to the benefits of automating the fault diagnosis process.

In the next three chapters, detection and classification models corresponding to the above mentioned three approaches are proposed and implemented. This includes a path-based model that utilizes data mining to find anomaly sequences in Chapter 4, a feature extraction model in Chapter 5, and two deep learning models in Chapter 6. Additionally, Chapter 7 addresses the use of timestamp information that is largely ignored by existing studies.

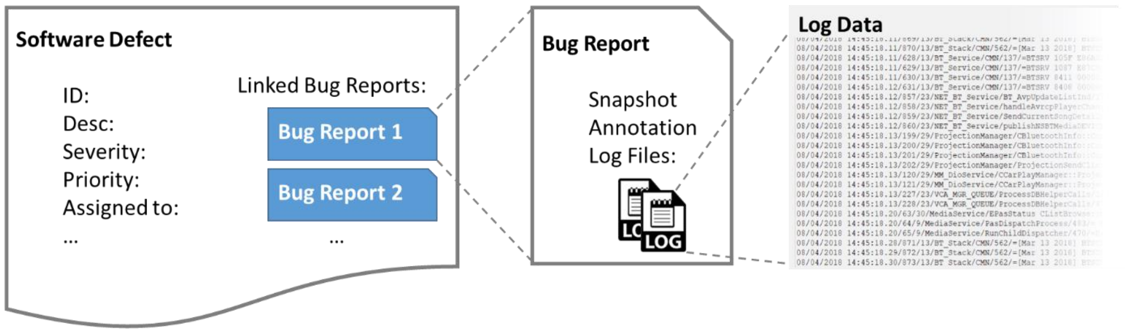
Chapter 4 Data Mining and Bayesian Classifier

This chapter proposes a method for Fault Diagnosis with Sequential Pattern Mining (FDSPM). In this method, a workflow-based model is used to automatically detect and diagnose software faults. This approach is based on the intuition that faulty software workflows produce distinguishing log patterns. The model's framework is similar to that of a typical machine learning model, comprising a learning or training phase and a prediction or classification phase. The learning phase uses data mining techniques to discover representative sequence patterns from historical faulty data. The classification phase produces detection and diagnosis results based on Bayesian probability theory. The model showed effectiveness in the evaluation using the Ford SYNC log dataset.

This chapter is organised as follows. Section 4.1 introduces the motivation of the work and the overview of the approach. Section 4.2 describes the processing needed to prepare sequence data. Sections 4.3 and 4.4 presents the pattern discovery algorithm and the classification method, respectively. Section 4.5 discusses the test preparation and Section 4.6 experiments with different model configurations. Section 4.7 concludes this chapter.

4.1 Motivation

The FDSPM is an attempt to mimic to the manual process of software defect triaging as described in Section 2.2. As a recap, triaging is the intermediate process between the discovery of a bug and the identification of a fault in the source code. The triaging process recognizes the fault from bug reports and categorizes it. The relationship among a software fault, bug report, and log data is shown in Figure 4.1.1. Given a bug report, the task of triaging is to determine whether it contains a fault and which fault it corresponds to through analyzing the log data.

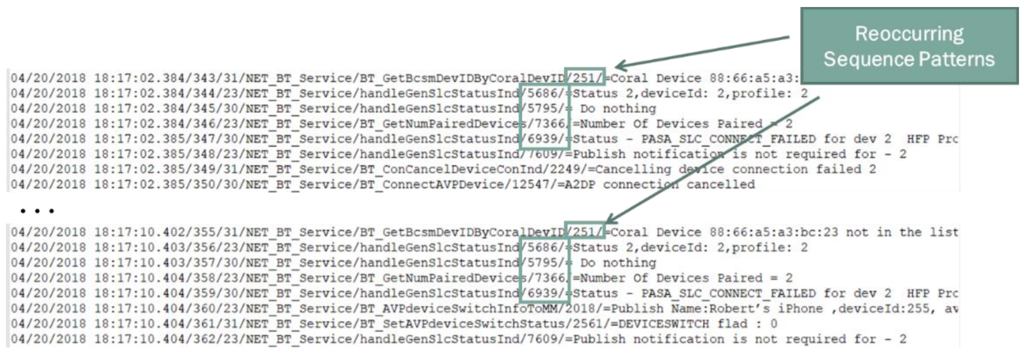


```
08/04/2018 14:45:18.11/869/13/BT_Stack/CMN/562/=[Mar 13 2018]
08/04/2018 14:45:18.11/870/13/BT_Stack/CMN/562/=[Mar 13 2018]
08/04/2018 14:45:18.11/628/13/BT_Service/CMN/137/=BTSRV 105F
08/04/2018 14:45:18.11/629/13/BT_Service/CMN/137/=BTSRV 1087
08/04/2018 14:45:18.11/630/13/BT_Service/CMN/137/=BTSRV 8411
08/04/2018 14:45:18.12/631/13/BT_Service/CMN/137/=BTSRV 8408
08/04/2018 14:45:18.12/857/23/NET_BT_Service/BT_AvpUpdateListInd
08/04/2018 14:45:18.12/858/23/NET_BT_Service/handleAvrcpPlayer
08/04/2018 14:45:18.12/859/23/NET_BT_Service/SendCurrentSongDetai
08/04/2018 14:45:18.12/860/23/NET_BT_Service/publishNSBTMedia
08/04/2018 14:45:18.13/199/29/ProjectionManager/CBluetoothInfo::
08/04/2018 14:45:18.13/200/29/ProjectionManager/CBluetoothInfo::
08/04/2018 14:45:18.13/201/29/ProjectionManager/CBluetoothInfo::
08/04/2018 14:45:18.13/202/29/ProjectionManager/ProjectionSend
08/04/2018 14:45:18.13/120/29/MM_DioService/CCarPlayManager::
08/04/2018 14:45:18.13/121/29/MM_DioService/CCarPlayManager::
08/04/2018 14:45:18.13/227/23/VCA_MGR_QUEUE/ProcessDBHelperCalls
08/04/2018 14:45:18.13/228/23/VCA_MGR_QUEUE/ProcessDBHelperCalls
08/04/2018 14:45:18.20/63/30/MediaService/EPasStatus CListBrowse
08/04/2018 14:45:18.20/64/9/MediaService/PasDispatchProcess/
08/04/2018 14:45:18.20/65/9/MediaService/RunChildDispatcher/470/
08/04/2018 14:45:18.28/871/13/BT_Stack/CMN/562/=[Mar 13 2018]
08/04/2018 14:45:18.29/872/13/BT_Stack/CMN/562/=[Mar 13 2018]
08/04/2018 14:45:18.30/873/13/BT_Stack/CMN/562/=[Mar 13 2018]
```

Figure 4.1.1 The anatomy of a software defect in the defect management system.

The manual triage process mainly relies on software testing know-how and experience in the system. According to software engineers in the triage team, keyword search is an effective way to pinpoint a failure, using words like “fatal” and “crash”. However, this simple technique only covers no more than one third of all bug report cases. More details must be uncovered through closely examining the log messages: reoccurring sequences of log messages are good indicators of many defects, especially when they rarely appear in regular program execution. The individual messages in these distinguishing sequences may appear benign, but a certain permutation of them should raise the investigator’s attention. Figure 4.1.2 shows a reoccurring pattern that indicates a Bluetooth connectivity fault in the SYNC system. The top and bottom blocks of log messages are from two different recordings of the same Bluetooth issue. The first five lines of each block are nearly identical. The "Line ID" field highlighted in the boxes further confirms the observation (various log message fields are detailed in Section 4.2). This repetition is one of the signature patterns for a Bluetooth fault and shows up in every log recording whenever the fault occurs.

The intuition is to directly obtain such sequence patterns and make use of them for diagnosis purposes. An *expert system* approach is to explicitly specify a set of patterns based on the knowledge of subject matter experts. But such an approach is shown to be expensive and cumbersome based on related literature reviewed in Chapter 2. The data-driven approach is to obtain these patterns from existing faulty log data that have been diagnosed. This process is referred to as *pattern discovery*. Each distinctive log line is represented by an identifier, referred to as a *token*. Pattern discovery is to find the token patterns that reoccur in different log sequence samples. The problem is called sequence mining in the data mining area.



```
04/20/2018 18:17:02.384/343/31/NET_BT_Service/BT_GetBcsmDevIDByCoralDevID/251/=Coral Device 88:66:a5:a3:
04/20/2018 18:17:02.384/344/23/NET_BT_Service/handleGenSlcStatusInd/5686/=Status 2,deviceId: 2,profile: 2
04/20/2018 18:17:02.384/345/30/NET_BT_Service/handleGenSlcStatusInd/5795/= Do nothing
04/20/2018 18:17:02.384/346/23/NET_BT_Service/BT_GetNumPairedDevices/7366/=Number Of Devices Paired = 2
04/20/2018 18:17:02.385/347/30/NET_BT_Service/handleGenSlcStatusInd/6939/=Status - PASA_SLC_CONNECT_FAILED for dev 2  HFP Prc
04/20/2018 18:17:02.385/348/23/NET_BT_Service/handleGenSlcStatusInd/7609/=Publish notification is not required for - 2
04/20/2018 18:17:02.385/349/31/NET_BT_Service/BT_ConCancelDeviceConInd/2249/=Cancelling device connection failed 2
04/20/2018 18:17:02.385/350/30/NET_BT_Service/BT_ConnectAVPDevice/12547/=A2DP connection cancelled
 • • •
04/20/2018 18:17:10.402/355/31/NET_BT_Service/BT_GetBcsmDevIDByCoralDevID/251/=Coral Device 88:66:a5:a3:bc:23 not in the list
04/20/2018 18:17:10.403/356/23/NET_BT_Service/handleGenSlcStatusInd/5686/=Status 2,deviceId: 2,profile: 2
04/20/2018 18:17:10.403/357/30/NET_BT_Service/handleGenSlcStatusInd/5795/= Do nothing
04/20/2018 18:17:10.404/358/23/NET_BT_Service/BT_GetNumPairedDevices/7366/=Number Of Devices Paired = 2
04/20/2018 18:17:10.404/359/30/NET_BT_Service/handleGenSlcStatusInd/6939/=Status - PASA_SLC_CONNECT_FAILED for dev 2  HFP Prc
04/20/2018 18:17:10.404/360/23/NET_BT_Service/BT_AVPdeviceSwitchInfoToMM/2018/=Publish Name:Robert's iPhone ,deviceId:255, av
04/20/2018 18:17:10.404/361/31/NET_BT_Service/BT_SetAVPdeviceSwitchStatus/2561/=DEVICESWITCH flad : 0
04/20/2018 18:17:10.404/362/23/NET_BT_Service/handleGenSlcStatusInd/7609/=Publish notification is not required for - 2
```

Figure 4.1.2 Example of reoccurring patterns in log data.

The patterns extracted can be used to match new log data for failure detection and fault diagnosis. Although it appears simple to determine a fault using the pattern-matching result, the problem becomes difficult when there are large numbers of patterns and faults. Often times there are multiple patterns extracted for each fault, denoted as $P_1, P_2, \ldots, P_m$, and multiple faults to diagnose, denoted as $h_1, h_2, \ldots, h_n$ (h for hypothesis). Using thresholds and rules to determine a possible fault quickly becomes complicated as m and n vary. A better way to approach this classification problem is to use Bayesian learning. Specifically, a Bayesian classifier can infer the most probable hypothesis h_j based on the occurrences of each pattern P_i in a given data sample. The requirement of applying Bayesian classifier is the prior knowledge of the conditional probability of every pattern under every fault, which can be obtained by observing the training data. Obtaining this set of prior knowledge, referred to as the *knowledge base*, is included in the learning process. In the diagnosis phase, a new log sample is pattern-matched by the extracted patterns, and the Bayesian classifier gives a classification result with a probability rating using the knowledge base.

This chapter proposes the FDSPM model that takes the sequence mining approach and addresses the pattern matching concern. Figure 4.1.3 shows the overview of FDSPM. The dashed line separates the training (learning) process from the detection (diagnosis) process. The learning process includes the following three modules.

- Data preprocessing: converts log data into token representation.

- Pattern discovery: extracts fault patterns using appropriate sequence mining algorithms.
- Knowledge base: obtains conditional probabilities for the discovered patterns.

The detection process shown below the dashed line is how the model performs fault diagnosis when deployed. Two important modules are included in this process.

- Pattern matching: examines a new log sequence against the sequence patterns from the pattern discovery process. If a match is found, the knowledge base gives the pattern's conditional probabilities.
- Bayesian classifier: produces a classification score (probability rating) for each fault using the pattern matching results and probabilities. The classifier then selects the fault with the highest score as the diagnosis output.

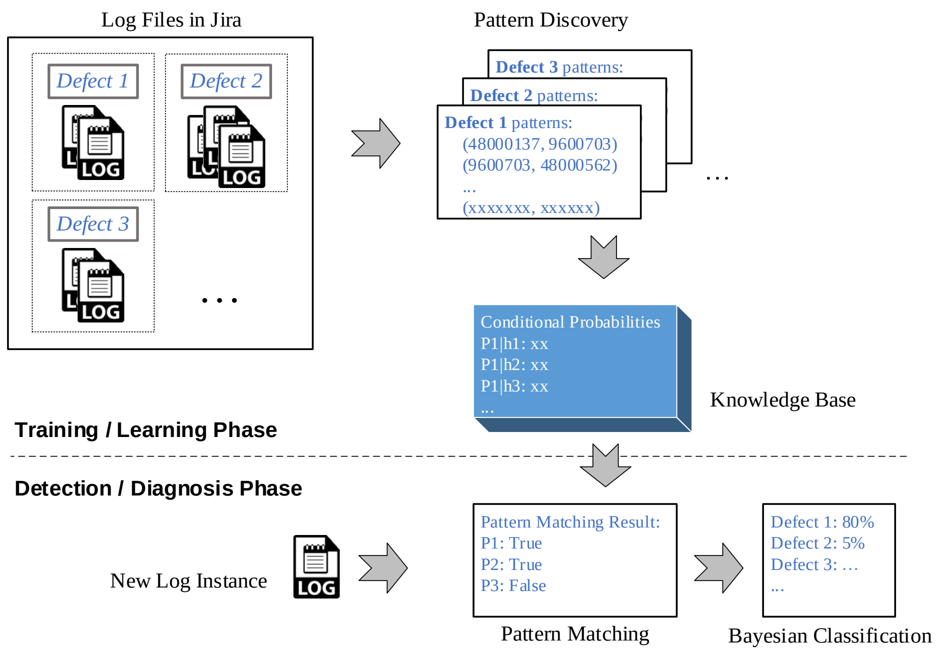


Figure 4.1.3. Overview of the FDSPM framework.

The processes in this framework are presented in the following subsections.

4.2Data Preparation

The FDSPM is a supervised learning task requiring labelled data for training and evaluation purposes. The log data and labels can be generated and are assumed to be available from a typical defect management system, which stores software defects, bug reports, and log data in the structure shown in Figure 4.1.1. The proposed FDSPM uses general nomenclature in machine learning: the term *log sample* represents a log file from the bug report, and a *label* or *class* represents the software defect ID.

Not all faults and log data stored in the defect management system are eligible or readily available for the training or testing process. Eligible data need to be selected, cleaned, and preprocessed for the following reasons.

- Training and evaluation require a minimum number of samples. For each fault, at least two log samples are required to capture the reoccurring patterns. This is only a bare minimum – as a pattern cannot reoccur with one sample. The evaluation requires at least one additional sample. As a result, a fault eligible for training and testing should contain a minimum of three bug reports. Faults that do not contain enough bug reports are not considered. This study has to work with a small number of samples due to a lack of available data. The intent is to demonstrate the effectiveness of the algorithms. More data samples than the bare minimum should be collected in an actual application for better results.

- Data cleaning is necessary. A vast majority of log messages within a log file are not related to the fault being analyzed. Removing unrelated log messages based on the fault category can improve the accuracy of pattern extraction and reduce the computational requirement.
- Log sequences need to be converted into token format, such that a pattern discovery algorithm can be performed.

The details of these data preparation steps are provided in the following subsections.

4.2.1 Data Selection and Cleaning

Because of the scale of the SYNC operating system and the time limitation of the study, it is not realistic to examine all defects in every module. The Bluetooth related log defects are selected in this study. The choice is made based on the fact that Bluetooth is one of the most troublesome modules and has the most defects and bug reports available in its associated defect management system.

Each eligible software defect should have at least three bug reports linked to it: two for training, one for testing. This requirement seems effortless to meet, as the testing vehicles and benches are producing log data all the time. In practice, however, the defect management system is designed to track the most valuable information, instead of storing a large quantity of data. As a result, amongst all Bluetooth-related faults associated with the Ford Sync system described in Section 2.2.3, only eight of them meet this criterion.

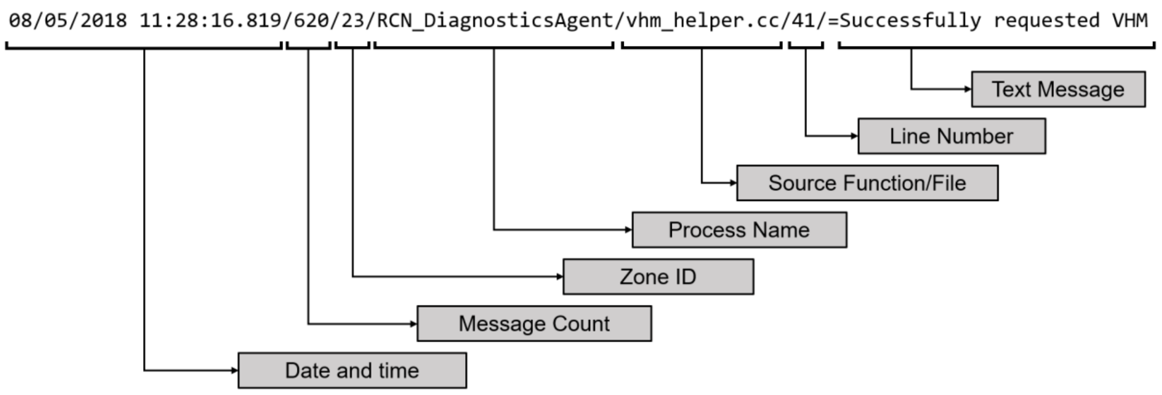
The details of eight selected defects and their 42 linked log samples are shown in Table 4.1. The “Defect ID” is assigned by the fault management system. The “Summary”

Table 4.1 The Bluetooth related defects and log data files. (Filenames are partially removed due to confidentiality)

Class Label	Defect ID	Summary	Filenames
1	FORDSYNC3-40557	LW_EVENT_PROCESS_CRASH in mm-ipod	QUIP_A0A815-------------------------- QUIP_7E8D25-------------------------- QUIP_8C6E2C-------------------------- QUIP_99FDA6-------------------------- QUIP_87B4AD-------------------------- QUIP_3B9EE6-------------------------- QUIP_057E38-------------------------- QUIP_1D6BD5--------------------------
2	FORDSYNC3-40026	Initiating BT pairing from Sync displays the pin on Sync but not on device	QUIP_9B39A7-------------------------- QUIP_7E37BD-------------------------- QUIP_2D2D28--------------------------
3	FORDSYNC3-38112	Blank screen appears on Phone screen when make an outgoing call	QUIP_2E42E5-------------------------- QUIP_8EEFD9-------------------------- QUIP_A710F3-------------------------- QUIP_A85045--------------------------
4	FORDSYNC3-37158	Voice command to make a phone call, throws error Contacts download not complete	QUIP_E72A76-------------------------- QUIP_8D3505-------------------------- QUIP_1C0491--------------------------
5	FORDSYNC3-32240	Wrong Bluetooth Streaming MetaData Displayed	QUIP_5F9359-------------------------- QUIP_7FC81A-------------------------- QUIP_52B417-------------------------- QUIP_209C9E--------------------------
6	FORDSYNC3-28906	Random Bluetooth disconnection and re-connection, Fatal Error:0x12.	QUIP_F0F926-------------------------- QUIP_4DCCAF-------------------------- QUIP_F088FA-------------------------- QUIP_957930--------------------------
7	FORDSYNC3-28578	Bluetooth did not connect automatically upon entering the car even though device BT is turned ON	QUIP_9A1E68-------------------------- QUIP_83272C-------------------------- QUIP_6C94AF-------------------------- QUIP_7DECAE-------------------------- QUIP_F67D03-------------------------- QUIP_D85B75-------------------------- QUIP_E30A9A-------------------------- QUIP_E30CDC--------------------------
8	FORDSYNC3-28414	Music went silent while playing BT audio and track timer continued	QUIP_6D76F5-------------------------- QUIP_3C01A6-------------------------- QUIP_7B38B3-------------------------- QUIP_3B8413-------------------------- QUIP_370925-------------------------- QUIP_0A9440-------------------------- QUIP_05A657-------------------------- QUIP_09D4D4--------------------------

“Filenames” column refers to the log data files. Note that each defect may have a different number of log samples. These data files are referred to as *faulty log samples*.

As introduced in Chapter 1, each log file contains ~70,000 log messages. A vast majority of them are unrelated to the defect being diagnosed. For example, when a Bluetooth failure is encountered, there is very little chance that it relates to the air conditioning module logs. Therefore, cleaning the log data based on the general category of the issue is unlikely to reduce the valuable information and can effectively reduce the size of log data. This is achieved by filtering the log messages by their “`Module Name`”, a structured field that exists in every log message as shown in Figure 4.2.1 (reiteration of Figure 2.3.1).



```
08/05/2018 11:28:16.819/620/23/RCN_DiagnosticsAgent/vhm_helper.cc/41/=Successfully requested VHM
```

Figure 4.2.1 The structure of one log message.

4.2.2 Tokenization

As mentioned in Chapter 3, extracting structured information from log data in its unstructured free-text format is called parsing. Specifically, in this path-based approach, parsing is to extract symbolic identifiers to represent log messages. This identifier is commonly referred to as a *token* and the process is called *tokenization*. Tokenization converts the log message sequence into a token sequence. Figure 4.2.2 shows an example of tokenizing a piece of Ford log messages sequence. Each line of log message is represented by a token as indicated on the left column. Note that the same log messages have the same token. After tokenization, this log sample is represented by the list: `[178801199, 48000562, 48000562, 48000137, …]`. Each integer in the list is an identifier rather than having an arithmetic meaning. The process of generating the token sequence is detailed in this section.

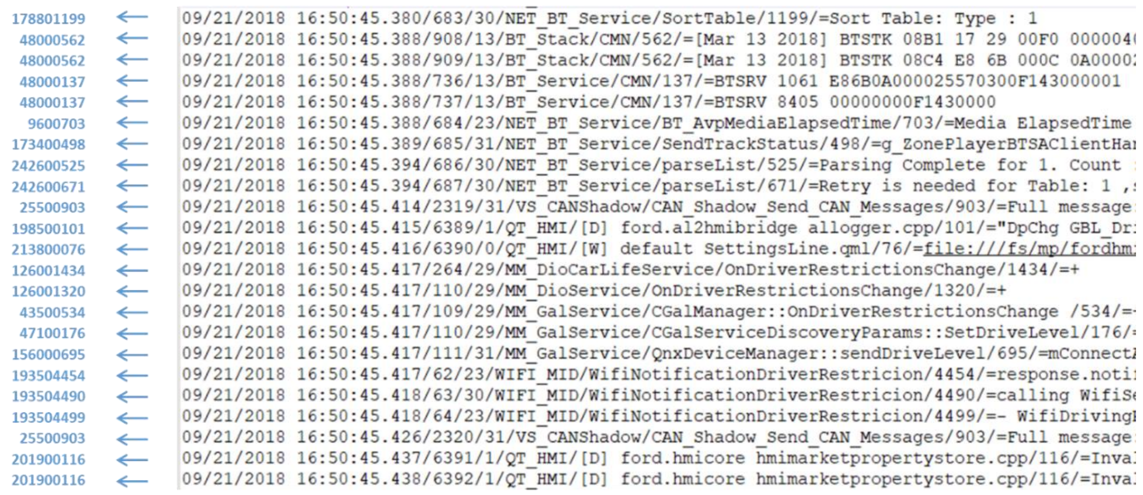
Parsing can be very sophisticated for some types of log data that have a completely unstructured format. Fortunately for the SYNC log, the structured fields within each log message provide enough information to identify individual logs. These fields are: `Process Name`; `Source Function`; and, `Line Number`, as previously shown in Figure 4.2.1. When a software *process* is running and the logging event happens, the log messages record the exact line of code being executed in the form of *source function* and *line number*. Therefore, a combination of process identifier, function identifier and the line number can uniquely represent each log message.

The available log data in this study contain a total of 3962 different (source) functions from all modules. A unique integer *function key* is assigned to each function. The values of function keys range from 0 to 3961. The line number is available from the log message; its value is always less than the total lines in a source code file. A single source file rarely exceeds 100,000 lines, so a range of 0 to 99,999 is sufficient to cover all possible line number values. Combining the function key and line number, the token value is produced using the following equation:

$$\text{TokenValue} = \text{FunctionKey} \times 100{,}000 + \text{LineNumber}$$

Generating and interpreting tokens are simple with this method. Take the first line of Figure 4.2.2 as an example, the `SortTable` function has a function key of 1788, and the message is from line number 1199, so the token for this message is 178801199. When interpreting a token, the last five digits represent the line number, and the higher digits represent the function key.

The mapping of function keys and function names is stored and updated as an external database, separated from the log tokenization process.



```
178801199  ←  09/21/2018 16:50:45.380/683/30/NET_BT_Service/SortTable/1199/=Sort Table: Type : 1
 48000562  ←  09/21/2018 16:50:45.388/908/13/BT_Stack/CMN/562/=[Mar 13 2018] BTSTK 08B1 17 29 00F0 000004
 48000562  ←  09/21/2018 16:50:45.388/909/13/BT_Stack/CMN/562/=[Mar 13 2018] BTSTK 08C4 E8 6B 000C 0A0000
 48000137  ←  09/21/2018 16:50:45.388/736/13/BT_Service/CMN/137/=BTSRV 1061 E86B0A000025570300F143000001
 48000137  ←  09/21/2018 16:50:45.388/737/13/BT_Service/CMN/137/=BTSRV 8405 00000000F1430000
  9600703  ←  09/21/2018 16:50:45.388/684/23/NET_BT_Service/BT_AvpMediaElapsedTime/703/=Media ElapsedTime
173400498  ←  09/21/2018 16:50:45.389/685/31/NET_BT_Service/SendTrackStatus/498/=g_ZonePlayerBTSAClientHa
242600525  ←  09/21/2018 16:50:45.394/686/30/NET_BT_Service/parseList/525/=Parsing Complete for 1. Count
242600671  ←  09/21/2018 16:50:45.394/687/30/NET_BT_Service/parseList/671/=Retry is needed for Table: 1 ,
 25500903  ←  09/21/2018 16:50:45.414/2319/31/VS_CANShadow/CAN_Shadow_Send_CAN_Messages/903/=Full message
198500101  ←  09/21/2018 16:50:45.415/6389/1/QT_HMI/[D] ford.al2hmibridge allogger.cpp/101/="DpChg GBL_Dr
213800076  ←  09/21/2018 16:50:45.416/6390/0/QT_HMI/[W] default SettingsLine.qml/76/=file:///fs/mp/fordhm
126001434  ←  09/21/2018 16:50:45.417/264/29/MM_DioCarLifeService/OnDriverRestrictionsChange/1434/=+
126001320  ←  09/21/2018 16:50:45.417/110/29/MM_DioService/OnDriverRestrictionsChange/1320/=+
 43500534  ←  09/21/2018 16:50:45.417/109/29/MM_GalService/CGalManager::OnDriverRestrictionsChange /534/=
 47100176  ←  09/21/2018 16:50:45.417/110/29/MM_GalService/CGalServiceDiscoveryParams::SetDriveLevel/176/
156000695  ←  09/21/2018 16:50:45.417/111/31/MM_GalService/QnxDeviceManager::sendDriveLevel/695/=mConnect
193504454  ←  09/21/2018 16:50:45.417/62/23/WIFI_MID/WifiNotificationDriverRestricion/4454/=response.noti
193504490  ←  09/21/2018 16:50:45.418/63/30/WIFI_MID/WifiNotificationDriverRestricion/4490/=calling WifiS
193504499  ←  09/21/2018 16:50:45.418/64/23/WIFI_MID/WifiNotificationDriverRestricion/4499/=- WifiDriving
 25500903  ←  09/21/2018 16:50:45.426/2320/31/VS_CANShadow/CAN_Shadow_Send_CAN_Messages/903/=Full message
201900116  ←  09/21/2018 16:50:45.437/6391/1/QT_HMI/[D] ford.hmicore hmimarketpropertystore.cpp/116/=Inva
201900116  ←  09/21/2018 16:50:45.438/6392/1/QT_HMI/[D] ford.hmicore hmimarketpropertystore.cpp/116/=Inva
```

Figure 4.2.2 Tokenization: converting log messages into tokens.

After cleaning and tokenization, each log file is converted into a token sequence sample using the process shown in Figure 4.2.2. After data selection, cleaning, and tokenization, the Bluetooth fault dataset comprises the sequence samples and their fault labels.

4.3 Pattern Discovery

Pattern discovery is a crucial process in the FDSPM framework. Specifically, it extracts a series of sequence patterns $P = [P_1, P_2, \ldots]$ from the log sequence samples $S = [S_1, S_2, \ldots]$ that contain the same fault. The sequence patterns P is also called *reoccurring patterns, common subsequences*, or *distinguishing patterns* of a fault. This intuition comes from the manual process of looking for unusual log patterns to infer a fault. A few facts need to be considered when searching sequence patterns P. First, the log messages often follow specific orders during fault propagation, as observed from the example shown previously in Figure 4.1.2 as well as the rest of the dataset. As a result, the sequence pattern P should preferably retain the order information. Secondly, the absolute positions of the patterns in the log sequence are not significant, as the fault can occur any time when the system is running. Lastly, the patterns are often short – no more than a few tokens – based on the observation of existing data. The length should be larger than or equal to 2, otherwise it becomes a keyword search problem.

4.3.1 Selecting a Mining Approach

Discovering patterns from a large dataset is the definition of data mining [77]. Specifically, given a set of sequence samples with the same label, data mining algorithms discover the reoccurring patterns that appear frequently in the set. There are two strategies for finding the reoccurring patterns: frequent pattern mining and contrast pattern mining.

4.3.1.1 Frequent Patterns

Take the following two sequences S_1 and S_2 as an example. A, B, and C represent the tokens that make up the sequences. In this simplified example, the evident longest common sequence pattern is AAC, because it appears in both sequences. Another common pattern is CB. If both S_1 and S_2 belong to the same class, then AAC and CB are likely the sequence pattern for this class.

$$S_1\texttt{: ...A A C A C B...}$$
$$S_2\texttt{: ...B A A C B A...}$$

This type of sequence pattern is usually seen in bioinformatics research [78], where the terminology *frequent pattern* [79] or *sequence motif* [80] is commonly used. An example is the Deoxyribonucleic Acid (DNA) genome, consisting of four nucleotides represented by the letters A, C, G, and T. Searching the frequent patterns from the genome sequences that have the same gene expression helps researchers understand and identify the interested genome.

There have been no studies investigating the frequent pattern mining techniques on log sequences applications. The bioinformatics-based methods theoretically work on the log data, but there are several limitations. First, the methodologies are mostly developed for biological sequences, where the elements (tokens) constructing the sequences are few. DNA nucleotides only have four types. Meanwhile, the software system log, such as the SYNC system, contain thousands of unique tokens. Secondly, the operating system is likely to multitask most of the time, creating interleaving log messages. This implies that when the sequence pattern shows up in the logs, it may be interrupted by other tokens and contain gaps in its presentation. Lastly and practically, the lack of available data (only a few samples for each fault) may have a negative effect on the accuracy of frequent pattern mining.

4.3.1.2 Contrast Data Mining

To address the lack of data issue, the *contrast data mining* approach can be adopted to incorporate log data from regular executions. Specifically, the normal system logs that contain no failures or faults are used as a *contrast set*. The data samples containing a fault are called a *target set*. The patterns extracted from the target set, i.e., the frequent patterns, may not be accurate and contain false patterns. These false patterns would be rejected if they also appear in the contrast set. As a result, the extracted patterns are more accurate in representing the fault.

In the following example, the two sequences S_1 and S_2 are in the target set containing a certain fault, while the sequences S_1' and S_2' are in the contrast set containing normal executions. The frequent patterns based on the target set are AAC and CB, as concluded in the previous section. However, since the pattern CB also appears in S_2' which occurs under a normal condition, it should not be considered as a fault pattern. Therefore, the correct pattern for this fault is AAC only. The patterns extracted using two opposing datasets like the ones shown in this example are called *distinguishing patterns* or *contrast patterns* [81]. The contrast patterns are a subset of – and more accurate than – the frequent patterns.

Target set:	Contrast set:
`...A A C A C B...`	`...B C A B...`
`...B A A C B A...`	`...A B A C B...`

The gap constraint is introduced to address the interleaving issue when extracting contrast patterns. This means that the pattern P may be interrupted by other tokens when appearing in a sequence. Take the previous example where $P_1 = \mathsf{AAC}$, it may show up in the form of A[*]A[*]C in a sequence, where [*] represents a gap that can be any tokens. A *g*-gap constraint specifies the number of tokens *g* allowed in the gap [*]. If *g* = 1, then the pattern ABABC is considered a valid appearance of P_1. Adding gap constraint will inevitably increase the computation complexity, and a larger *g* is likely requiring more computation. Note that previously in the log data selection process, only Bluetooth related modules are chosen. The interleaving effect is likely minimized during the selection process. Nevertheless, it is worth investigating the gap constraint to confirm whether it affects the detection or diagnosis performance.

In either the frequent or contrast mining approach, a pattern may not be perfectly showing up in every sample of the dataset, so a threshold is required to determine if a pattern should qualify as a “common pattern”. The *support count* of a pattern P is defined as how many samples in the sequence set $[S_1, S_2, \ldots]$ match the pattern P. *Support ratio* is the support count divided by the total number of samples. For frequent mining, a Support Ratio Threshold (SRT) is often manually specified. A pattern P’s support ratio must be greater than the SRT in order to qualify as a common pattern. In the contrast mining case, there are two support ratios, i.e., the ratio of P in the target set and the ratio of P in the contrast set. As a result, two SRTs are specified. An eligible contrast pattern *P* must satisfy: 1) *P*'s support ratio in the target set is greater than a *minimum* SRT and, 2) *P*'s support ratio in the contrast set is less than a *maximum* SRT. The effects of both SRTs are investigated in the experiments section.

4.3.2 Contrast Data Mining with Gap Constraint

One particular data mining algorithm, the ConSGapMiner [82], addresses the two considerations above. The ConSGapMiner is short for Contrast Sequences with Gap Miner. It is relatively efficient considering its capability of incorporating gap constraint. The algorithm is briefly introduced as follows.

- **The framework.** The algorithm performs a depth-first search using a prefix tree as shown in Figure 4.3.1. The tree starts with a root node of NULL, as indicated by the curly brackets {} in the figure. Every other node on the branches represents an element from the token alphabet. The tree grows in lexicographic order one branch at a time. A pattern *P* is represented by traversing from the root node to a leaf node.
- **Growing the tree.** When adding a node, a new pattern is generated. Its support ratios are calculated and stored at the node. Two SRTs – the minimum SRT and the maximum SRT – between 0 and 1 are set to determine whether a pattern should be qualified as a contrast pattern. If the new pattern satisfies both ratio thresholds, it is selected as a candidate pattern and the branch stops growing. In a deterministic setting, the minimum threshold for the target set is 1.0, meaning all

samples in the target set must contain the pattern P, and the maximum threshold for the contrast set is 0.0, meaning the pattern P cannot appear in the contrast set.

- **Efficiency measures.** The most computationally intensive process is the calculation of the two support ratios when a new node is created. The ConSGapMiner algorithm uniquely utilizes a *bitset array* representation to describe and calculate the occurrences of one sequence within another. For example, the bitset representation of a pattern $P = \texttt{AB}$ occurring in sequence $S = \texttt{BACACBCCB}$ is a binary code `000001001`, which has the same length as S. In the binary code, a "`1`" is turned on at the final position where the pattern `AB` could be embedded (this example considers the gap constraint). The efficiency improvement mainly attributes to turning the sequence searching operation into binary shifts and logical operations, while addressing the gap constraint at the same time.
- **Other optimization techniques**. The maximum depths can be set manually to prevent irrationally long patterns and reduce computation time. The algorithm also comes with an early stopping mechanism by pruning the branches.

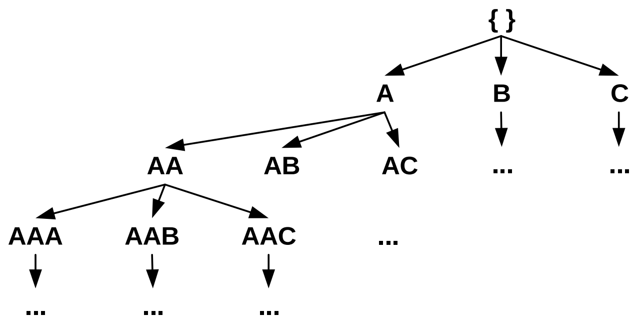


Figure 4.3.1 The prefix tree structure used in contrast data mining. (alphabet = {A,B,C})

A detailed explanation of the algorithm can be found in Chapter 6 of Dong and Pei [81]. The pseudo-code referred from the source can be found in the Appendix.

4.3.3 Implementation

The log data cleaning, tokenization, and pattern mining algorithm are implemented using Python on Windows 10. The source code for ConSGapMiner is not available from the original publication, so a Python version is implemented based on the pseudo-code provided with the paper. In the implementation, the gap constraint, maximum and minimum support thresholds, and maximum pattern length are configurable.

An overview of the pattern discovery process is illustrated in Figure 4.3.2. The process is carried out one defect (fault label) at a time. The tokenization process first converts the faulty log files into token sequence samples. These are samples that contain faults, so they are the target set for the pattern mining algorithm. The contrast set is created similarly using the regular log data. Then the contrast sequence mining algorithm finds the feature patterns $P = [P_1, P_2, \dots]$ using the target and contrast sets. The process is carried out for all the defects and produces a list of contrast patterns for each defect listed in Table 4.1. These extracted patterns of each defect should be different, unless two defects are closely related or exhibit similar behaviour.

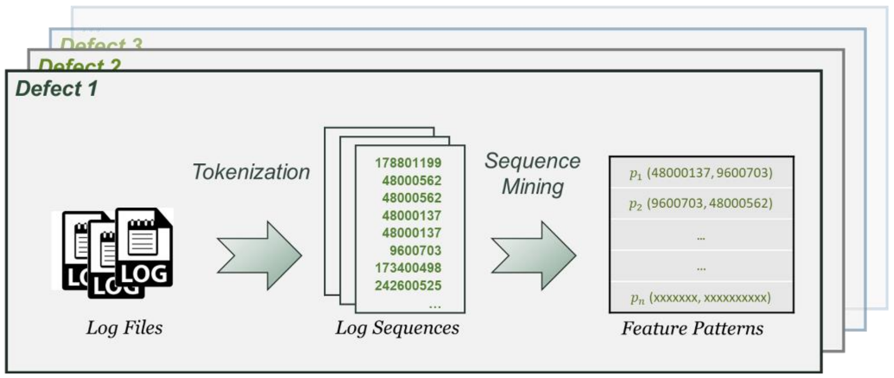


Figure 4.3.2 The workflow of feature pattern discovery

Ideally, the contrast set should be made of regular execution log files. Log data are constantly generated under normal conditions, but collecting them turns out to be a problem. Normally the data are collected using the defect management system described in Section 2.2, in the form of a bug report. However, collecting a large amount of normal data in this way would flood the bug report system with non-fault-related items, hindering the software teams' daily work. A workaround is to use existing data in the defect management system. In the management system, some bug reports are determined as "Expected behavior", "Incomplete", or "Won't fix" after triaging. This means that the developers could not find enough evidence to prove a defect exists, or could find minor defects but which are not worth fixing. Although these data are not perfectly error-free, they could be used as the contrast set as they contain no clear patterns recognizable by the human experts. In this study, they are treated as *normal log samples*. In total, 72 normal log samples are uncovered. It is worth noting that the use of exiting non-fault data is only a workaround. Using a comprehensive healthy log dataset as the contrast set is the best practice and likely to produce more accurate contrast patterns. However, due to the limitation of resources and access to the data, constructing such a dataset is difficult within a reasonable timeframe, if feasible at all.

4.4 Fault Classification

After the pattern discovery process, a list of contrast patterns is obtained for every fault. To diagnose a new log sample, these contrast patterns are used to pattern-match the new sequence. A fault can be determined if a pattern shows a match. However, this simplistic approach has several drawbacks. First, the appearance of a fault may vary under different operating conditions. Concretely, pattern discovery for fault 1 produces a number of contrast patterns $P = [P_1, P_2, \dots]$, but not all of which appear in every sample of fault 1. In other words, a subset of P may be sufficient to determine fault 1 in a log sample. Secondly, the extracted patterns may not be accurate or complete, due to the lack of comprehensiveness of available datasets. Additionally, each fault has a different number of patterns, determining a pass-fail threshold for the pattern-matching can be tricky. Therefore, directly using the pattern-matching result to determine a fault can be ineffective in this multi-class, multi-pattern setting. The FDSPM utilizes the Bayesian approach to tackle this classification problem.

4.4.1 Naïve Bayes Classifier

In machine learning, Bayesian learning is a probabilistic approach to infer classification. In this diagnosis scenario, denote a log sample as D and a set of hypotheses as $H = [h_1, h_2, \dots, h_n]$, each h_i represents a possible fault, the task is to infer the maximally probable hypothesis, or *maximum a posteriori* (MAP) hypothesis by observing data D:

$$h_{MAP} = \underset{h_j \in H}{\operatorname{argmax}} P(h_j | D) \tag{4.1}$$

The log sample D can be represented by the pattern-matching result $p = [p_1, p_2, \dots, p_m]$, where p_i denotes whether the pattern P_i occurs or not. This process of examining feature patterns in log sample D is referred to as *observation* in statistics. Note that the observation result p is in lowercase, and fault patterns P is in uppercase. While the expression $P(\ )$ with brackets represents a probability function. The MAP definition can be written as:

$$h_{MAP} = \underset{h_j \in H}{\operatorname{argmax}} P(h_j | p_1, p_2, \dots, p_m) \tag{4.2}$$

With the Bayes theorem $P(h|D) = \frac{P(D|h)P(h)}{P(D)}$, the MAP hypothesis of the log pattern matching is represented as follows:

$$h_{MAP} = \underset{h_j \in H}{\operatorname{argmax}} \frac{P(p_1, p_2, \dots, p_m | h_j) P(h_j)}{P(p_1, p_2, \dots, p_m)} \tag{4.3}$$

Here $P(p_1, p_2, \dots, p_m)$ represents the observation result of all patterns $[P_1, P_2, \dots P_m]$ in the given log sample D. Clearly this value remains constant because the log sample being examined do not change, so the denominator can be omitted:

$$h_{MAP} = \underset{h_j \in H}{\operatorname{argmax}} P(p_1, p_2, \dots, p_m | h_j) P(h_j) \tag{4.4}$$

In this representation, the term $P(p_1, p_2, \dots, p_m | h_j)$ represents the probability of a pattern-matching combination $[p_1, p_2, \dots, p_m]$ given the hypothesis h_j holds, i.e., the *j*-th fault occurs. The term $P(h_j)$ represents the unconditional probability of the *j*-th fault.

The value of $P(p_1, p_2, \dots, p_m | h_j)$ needs to be obtained from the training data, i.e. the data used for the pattern extraction process. However, explicitly finding the condition $p_1, p_2, \dots, p_m | h_j$ is infeasible, because the available log samples cannot possibly cover the huge number of possible combinations $[p_1, p_2, \dots, p_m]$ in any way. Assumptions need to be made to proceed with the deduction and, result in different methods, one of them being the naïve Bayes classifier.

The naïve Bayes classifier assumes that all feature attributions – in this case, the occurrences of extracted patterns $p_1, p_2, \dots, p_m$ – are conditionally independent given the target hypothesis h_j. As a result, the probability of observing the combination $p_1, p_2, \dots, p_m$ equals the product of the probabilities of observing individual patterns: $P(p_1, p_2, \dots, p_m | h_j) = \prod_{i=0}^{m} P(p_i | h_j)$. The MAP hypothesis then becomes naïve Bayes classifier:

$$h_{NB} = \underset{h_j \in H}{\operatorname{argmax}} P(h_j) \prod_{i=0}^{m} P(p_i | h_j) \tag{4.5}$$

Where $P(p_i | h_j)$ is the conditional probability, and $P(h_j)$ is called prior probability.

4.4.2 Learning and Classification

In equation (4.5), given a piece of log sequence and the observed patterns $p = [p_1, p_2, \dots, p_m]$, $P(h_j)$ and $P(p_i | h_j)$ are required to obtain a fault classification. $P(h_j)$ is the unconditional probability of the fault j occurring. This can be approximated by the frequency of fault j occurring in the available data, i.e., the number of samples with fault j among all faulty samples. $P(p_i | h_j)$ is the conditional probability of pattern p_i when fault j occurs. This again can be obtained from the training data: it equals the frequency of observing a pattern p_i given a fault h_j holds. Since a pattern does not necessarily appear in every fault, $P(p_i|h_j)$ has a value between 0 and 1.

The obtained $P(p_i|h_j)$ commonly has zero values, because some patterns do not show up at all in the data samples of some faults. For example, the pattern P_k is one of the extracted sequence patterns for fault k, and it does not occur in any data samples for the fault h_q. Then $P(p_k|h_q) = 0$. In the classification process using equation (4.5), if P_k is observed in a log sequence, it will bring the sum of product $\prod_{i=0}^{m} P(p_i|h_q)$ directly to zero, giving a zero probability for fault h_q, even though there may exist other patterns that contributes to h_q. This is likely caused by the limited amount and variety of the training data, from which the extraction of pattern P_k was unsuccessful or imprecise. In practice, to avoid the flushed-by-zero problem, the zero values of $P(p_i|h_q)$ are replaced with an arbitrarily small value, such as 0.1.

Given m features and n hypothesis, the set of conditional probabilities $P(p_i|h_j)$ is in matrix form as shown in Figure 4.4.1. The matrix has m rows and n columns, and each entry (i, j) corresponds to $P(p_i|h_j)$. It is the learnt result from training data, and is therefore referred to as knowledge base in Figure 4.1.3. The probability matrix and the prior probabilities are sufficient to calculate the classification result using equation (4.5), given the pattern-matching observation of a new log sample.

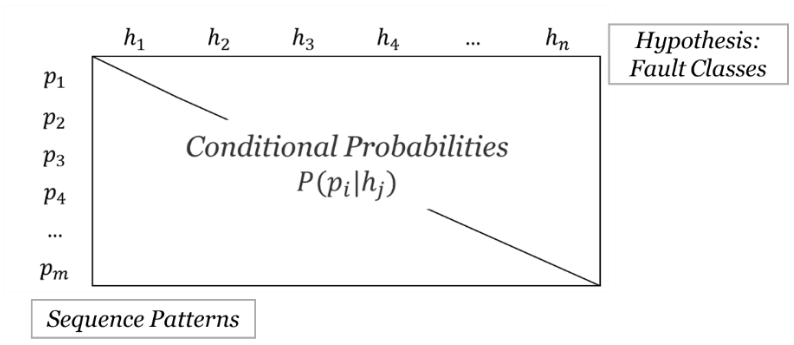

h_1 h_2 h_3 h_4 ... h_n
Hypothesis: Fault Classes
p_1 p_2 p_3 p_4 ... p_m
Conditional Probabilities $P(p_i|h_j)$
Sequence Patterns

Figure 4.4.1 The conditional probabilities matrix (knowledge base).

In the detection/diagnosis process, the pattern matching results $[p_1, p_2, \dots p_m]$ are obtained from observing a new log sequence. The conditional probabilities $P(p_i|h_j)$ are looked up from the knowledge base. The classification result is then obtained by equation (4.5).

4.4.3 Discussion on the Validity of the Assumption

The assumption that all extracted sequence patterns are conditionally independent given a fault barely holds. It is very much possible that two abnormal patterns always occur together, and one pattern triggers the logging action of the other. The contrast data mining is by no means to remove such dependencies. In fact, many naïve Bayes classification applications proceed with this inaccurate assumption, such as text classification [10]. Somewhat surprisingly, the naïve Bayes learners perform reasonably well in these applications as well as in this study, despite the inaccuracy of this assumption. Domingos and Pazzani [83] investigated this phenomenon and found that a naïve Bayes classifier “does not depend on attribute independence to be optimal”, which explains its exceptional performance despite the inaccurate assumption.

4.5 Implementation

A total of 104 log samples are collected for the experiment. This includes 42 faulty log samples as listed in Table 4.1 in Section 4.2.1 and 72 normal log samples as explained in Section 4.3.3. The normal samples are labelled as 0 and the faults are labelled from 1 to 8 as shown in Table 4.2. The data samples are split into training and testing set at a ratio of 0.5:0.5. Note that some fault labels contain an odd number of samples, and as a result, the training set would contain one sample more than the testing set due to rounding. After splitting, the number of samples for training and testing, the associated division is 58 and 56, respectively.

Table 4.2 The available data and train-test split.

Class Label	Fault ID	Total Samples	Samples for Training	Samples for Testing
0	No Identifiable Fault	72	36	36
1	FORDSYNC3-40557	8	4	4
2	FORDSYNC3-40026	3	2	1
3	FORDSYNC3-38112	4	2	2
4	FORDSYNC3-37158	3	2	1
5	FORDSYNC3-32240	4	2	2
6	FORDSYNC3-28906	4	2	2
7	FORDSYNC3-28578	8	4	4
8	FORDSYNC3-28414	8	4	4
	Total	114	58	56

The implementation flowchart of the proposed FDSPM framework is depicted in Figure 4.5.1, where the training and testing phases are shown on the left and right, respectively. The inputs to the training phase are the normal samples with label 0 in Table 4.2 and the ones with a certain fault label h_i. They are used as the contrast and target sets for the pattern discovery process. Each fault label is processed separately, so a total of 8 pattern discovery processes are carried out. All discovered feature patterns are combined in one list $[P_1, P_2, \dots, P_m]$. The conditional probabilities matrix $P(p_i|h_j)$ and the prior probability $P(h_j)$ are obtained by observing the frequencies of patterns in the training data.

In the testing phase, a pattern-matching process is first carried out to determine whether a pattern is observed. Given a new sample of token sequence, every extracted pattern in the list $[P_1, P_2, \dots, P_m]$ is searched within the sample. If a match is found, the matching result p_i is set to 1, otherwise, it is set to 0. The pattern-matching results are represented as a list $[p_1, p_2, \dots, p_m]$. Then the conditional probability $P(p_i|h_j)$ of an observed pattern p_i is looked up from the knowledge base matrix as previously shown in Figure 4.1.3. Using the naïve Bayes classifier represented in equation (4.5), the classification results and their probabilities are obtained.

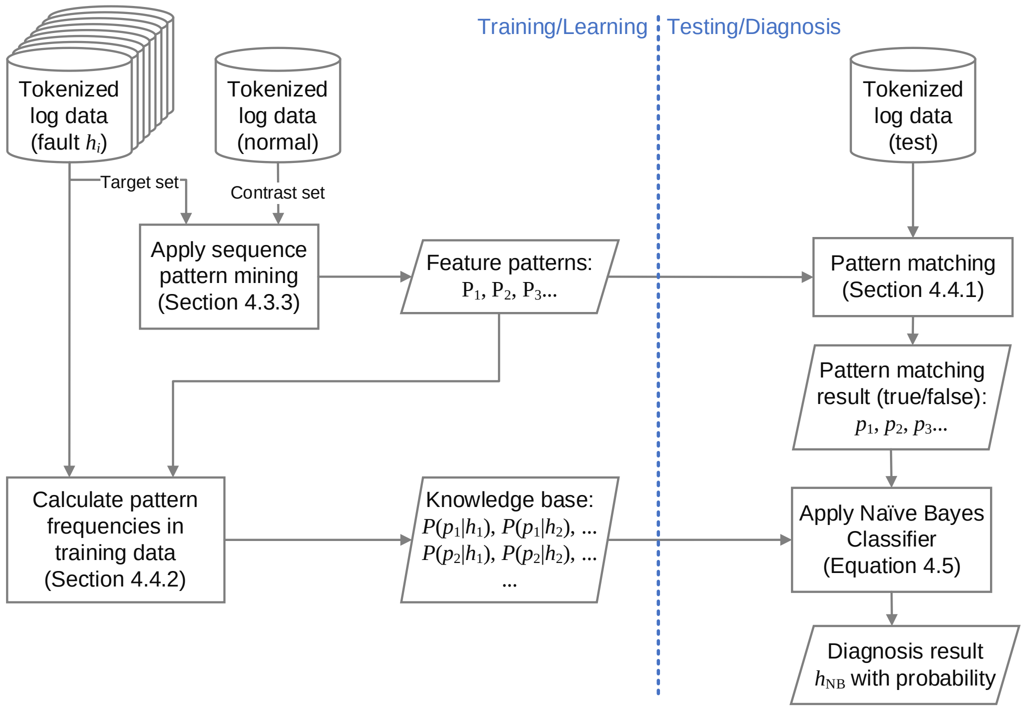

Training/Learning
Testing/Diagnosis
Tokenized log data (fault h_i)
Tokenized log data (normal)
Tokenized log data (test)
Target set
Contrast set
Apply sequence pattern mining (Section 4.3.3)
Feature patterns: P_1, P_2, P_3...
Pattern matching (Section 4.4.1)
Pattern matching result (true/false): p_1, p_2, p_3...
Calculate pattern frequencies in training data (Section 4.4.2)
Knowledge base: $P(p_1|h_1), P(p_1|h_2)$, ... $P(p_2|h_1), P(p_2|h_2)$, ...
Apply Naïve Bayes Classifier (Equation 4.5)
Diagnosis result h_{NB} with probability

Figure 4.5.1 The FDSPM model flowchart.

A few parameters are configurable and remain to be investigated in this model. These include the maximum feature pattern length, whether gaps are allowed in the feature patterns, the occurrence counts of feature patterns, and the probability threshold to determine whether and output should be considered faulty. The effects of these parameters are discussed in the experiments section.

The program is implemented in Python 3.6 on Windows 10. The link to the source code is listed in Appendix B. The hardware to run the experiments includes an Intel i7 processor and 32GB memory. This set of hardware is fully capable in terms of performing the calculations. The contrast pattern mining is computationally the heaviest process. Once the patterns are obtained, calculating conditional probabilities and classification are relatively fast, requiring minimal computation time and resources.

4.6 Experiments

The parameters configured during the data mining process can largely affect the quality of extracted patterns and the classification accuracy. These parameters are introduced in Section 4.3 and are reiterated as follows:

- **the gap constraint** specifies whether the patterns are allowed to have gaps when extracted from the sequence samples and the maximum gap allowed. Allowing patterns to have gaps increases the tolerance for interleaving sequences, but may also introduce false positive patterns.
- **the maximum length** configures the longest pattern length during the extraction.

- **the minimum and maximum Support Ratio Thresholds (SRT)** specify the criteria of eligible contrast patterns as introduced in Section 4.3.1. The pattern must occur in the target dataset with a support ratio higher than the minimum SRT and occur in the contrast dataset with a support ratio lower than the maximum SRT in order to be considered as an eligible contrast pattern. Setting strict SRTs (values close to 0 or 1) improves the quality of extracted patterns, but also reduces the total number of patterns extracted.

Another parameter to be manually configured is the **probability threshold** during the classification process. The naïve Bayes classifier produces the most likely class with a probable rating, but the output classes exclude label 0, i.e., the classifier does not detect normal samples that contain no faults. In practice, an arbitrary probability threshold is set to perform the detection functionality. If the most likely class has a probable rating lower than the probability threshold, the sample is considered to be normal with a class label 0.

The effects of varying these parameters are examined in a series of tests elaborated in this section. To produce consistent results, all tests use the same train-test split setting. As mentioned in Table 4.2, the testing set contains 36 normal samples with label 0 and 20 faulty samples with 8 other labels. Three metrics used in evaluation are: 1) the detection accuracy on faulty samples, also called True Positive rate (TP rate), recall, or sensitivity; 2) the detection accuracy on normal samples, also called True Negative rate (TN rate) or specificity; 3) the classification accuracy on faulty samples. The first two metrics are for evaluating detection performance, while the third one is for diagnosis.

4.6.1 The Gap Constraint

This set of tests specifies a series of gap values during the pattern extraction phase. The whole training and testing process, including pattern extraction, probability calculation, and classification, are individually performed for each test. The train-test data split and all other parameters are kept constant throughout this set of tests. The mining parameters are: maximum SRT=0.05, minimum SRT=0.6, maximum length=2. Note that these values are empirical values that yield reasonable results and by no means are intended to be optimal. The purpose of this set of tests is to examine what effect the gap constraint would impose.

With different gap settings, the extracted sequence patterns differ. This difference is shown in Table 4.3. In general, more patterns are extracted as the gap constraint increases, as demonstrated by the larger numbers in the bottom rows of the table. This is reasonable because a larger gap constraint allows a pattern to be present with a higher variety. The only outlier of this observation is fault 2, whose number of patterns does not change in the same way as the gap does.

Table 4.3 Number of extracted patterns with different gap settings.*

		Number of patterns extracted for each fault label								
Test #		fault 1	fault 2	fault 3	fault 4	fault 5	fault 6	fault 7	fault 8	Total
1	*gap*=0	15	0	15	6	1	18	2	0	57
2	*gap*=1	27	1	27	11	2	32	5	2	107
3	*gap*=2	36	1	42	14	3	51	6	4	157
4	*gap*=3	49	0	52	17	5	58	7	4	192
5	*gap*=4	59	1	57	22	9	70	11	5	234
6	*gap*=5	77	0	67	24	13	83	11	6	281

**All tests in this table use these mining parameters: maximum SRT=0.05, minimum SRT=0.6, maximum length=2.*

The average processing time to extract patterns for a fault is about two minutes using the performance PC mentioned above. The computation time increases by 5% to 10% for every increment of the gap setting. However, the memory usage increases at a faster rate and relates to the total number of patterns being extracted.

Table 4.4 shows the detection and classification accuracies of different gap settings. The detection accuracy column is evaluated on all test samples, including both normal and faulty ones. It is effectively the combination of true positive rate and true negative rate. The classification accuracy is evaluated on 20 faulty samples with 8 different labels – therefore the percentage figures have an increment of 5%. According to this table, both metrics show no distinctively correlation with the increase of gap values. The best-performing settings in terms of detection are gap-4 followed by gap-0, while in terms of classification, the winning configurations are gap-3 and gap-5.

Based on this observation, larger gaps tend to produce more sequence patterns, but the contribution of the extra patterns is not always beneficial, sometimes they even lower the performance.

Table 4.4 The performance with different gap settings.

Test #		Detection accuracies on all samples	Classification accuracies on faulty samples
1	*gap* = 0	62.5%	45%
2	*gap* = 1	60.7%	45%
3	*gap* = 2	58.9%	35%
4	*gap* = 3	55.4%	50%
5	*gap* = 4	64.3%	45%
6	*gap* = 5	57.1%	50%

4.6.2 The Maximum Length

The effect of patterns' length is examined in this set of tests. The maximum length parameter is specified in the pattern extraction process. This value can be as small as 1, but then it would become a simple keyword search problem. Therefore, the selection of lengths starts from 2. Table 4.5 shows the number of patterns extracted for each fault using different length settings. The mining parameters for this set of tests are: maximum SRT=0.05, minimum SRT=0.6, gap=0. Again, these values do not mean to represent the optimal setting.

According to this table, the number of patterns for each fault always increase in the same way as the maximum length does, and the increment becomes smaller for larger length values. Another important observation is that the sequence patterns for some faults are always greater than 2. Take fault 2 as an example, no pattern is found with a length of 2, but with a length of 3, one pattern is extracted. It means that the only sequence pattern for fault 2 has a length of 3. This finding is significant, indicating that a small length setting could miss valuable information during the pattern extraction process.

The monotonical increment of patterns can be explained based on the algorithm. In the mining algorithm presented in Section 4.3.2, the maximum length equals the depth of the search tree. If all other parameters are kept constant, the tree would always grow in the same way. Therefore, the patterns extracted using a larger length setting automatically include the ones extracted using a smaller setting.

Table 4.5 Number of extracted patterns with different length settings.*

Test #	*Max Length*	**Number of patterns extracted for each fault label**								
		fault 1	fault 2	fault 3	fault 4	fault 5	fault 6	fault 7	fault 8	Total
7	*2*	15	0	15	6	1	18	2	0	57
8	*3*	16	1	19	6	2	21	3	0	68
9	*4*	21	1	20	6	2	22	3	0	75
10	*5*	22	1	20	6	2	22	4	0	77
11	*6*	22	1	20	6	2	22	4	0	77
12	*7*	22	1	20	6	2	23	4	1	79

**All tests in this table use these mining parameters: maximum SRT=0.05, minimum SRT=0.6, gap=0.*

The processing time increases very little as the maximum length is set higher, and the memory usage varies in a small range. This can be explained from the search tree perspective, as the tree grows, a majority of branches fail the contrast pattern thresholds and therefore are pruned. The growth only happens at a small number of branches. As a result, the computation and memory increment are insignificant.

Table 4.6 shows the detection and classification accuracies using different length settings. The detection accuracy has a positive correlation with the maximum length parameter, or equivalently, the total number of extracted patterns. It shows that the additional longer patterns – especially patterns having a length of 3 and 4 – have a positive effect on the detection performance. In terms of classification, this effect is less obvious. The better performing length setting is 3, 7, and 8. Considering both metrics, a maximum length of 7 or 8 is the best choice, followed by a value of 3.

As a short conclusion for this section, this set of tests demonstrates that the sequence patterns for some faults are longer than 2. These longer patterns have a positive contribution to the detection performance. This means that when choosing an effective mining configuration, a higher maximum length is preferred.

Table 4.6 The performance with different length settings.

Test #		Detection accuracies on all samples	Classification accuracies on faulty samples
7	*Max length* = 2	69.6%	45%
8	*Max length* = 3	71.4%	50%
9	*Max length* = 4	73.2%	40%
10	*Max length* = 5	73.2%	40%
11	*Max length* = 6	73.2%	40%
12	*Max length* = 7	73.2%	50%
13	*Max length* = 8	73.2%	50%

4.6.3 The Support Ratio Thresholds

The maximum and minimum SRTs are examined together in this set of tests. The minimum SRT is used to determine if a pattern occurs frequently enough in the training data to be considered as a common pattern. The minimum SRT is relatively easy to configure, because a majority of faults only have two training samples, as indicated in Table 4.2. This means that a pattern's support in training samples, i.e., the pattern's occurrence count, is either 1 or 2. Then the minimum support ratio is either 0.5 or 1.0. The maximum SRT is used to determine if a pattern occurs too much in contrast set that it should no longer be considered as a distinguishing pattern. Since there are a lot more samples in the contrast set, i.e., the samples containing no faults, the maximum SRT values are selected empirically. The range from 0 to 0.25 is put into test, with an increment of 0.5. In total there are 12 combinations of selected maximum and minimum SRTs.

Table 4.7 shows the number of extracted patterns with different SRT combinations. Note the first 6 rows (tests #14-19) use a minimum SRT value of 0.6, which is effectively equivalent to 0.5 because the support count is rounded to integer values. All 12 tests in the table uses the mining parameters of maximum length=7 and gap=0, which are the preferred values as the previous subsection has concluded. According to this table, a smaller maximum SRT produces fewer patterns under each fault, while a smaller minimum SRT has the opposite effect. This is expected as a small maximum SRT close to 0.0 and a large minimum SRT close to 1.0 both means stricter thresholds to determine eligible contrast patterns.

Table 4.7 Number of extracted patterns with different SRT settings.

Test #	*Min SRT*	*Max SRT*	**Number of patterns extracted for each fault label**								
			fault 1	fault 2	fault 3	fault 4	fault 5	fault 6	fault 7	fault 8	Total
14	*0.6*	*0.0*	1	0	11	0	1	12	0	0	25
15	*0.6*	*0.05*	16	1	19	6	2	21	3	0	68
16	*0.6*	*0.1*	73	1	31	14	17	43	8	11	198
17	*0.6*	*0.15*	84	3	47	17	20	75	9	23	278
18	*0.6*	*0.2*	110	4	59	21	20	85	12	26	337
19	*0.6*	*0.25*	122	4	62	26	22	89	14	31	370
20	*1.0*	*0.0*	0	0	11	0	1	12	0	0	24
21	*1.0*	*0.05*	0	1	19	6	2	21	0	0	49
22	*1.0*	*0.1*	31	1	31	14	17	43	0	0	137
23	*1.0*	*0.15*	37	3	47	17	20	75	0	1	200
24	*1.0*	*0.2*	48	4	59	21	20	85	0	1	238
25	*1.0*	*0.25*	53	4	62	26	22	89	0	1	257

**All tests in this table use these mining parameters: maximum length=7, gap=0.*

In terms of computation, the training time of the 12 tests shows no significant difference. The memory usage is related to the number of patterns being extracted, but the total usage is within the hardware's capability in all cases.

Table 4.8 shows the performance difference among tests #14-25. As minimum SRT fixed and maximum SRT increases, the detection and classification accuracies generally show an upward trend, then settle or decrease. The highest detection accuracy occurs at maximum SRT=0.1, and the highest classification accuracy occurs at maximum SRT=0.2, regardless of the value of minimum SRT. A minimum SRT value of 0.6 seems to exhibit a smaller variance in terms of accuracy values. Other than that, the minimum SRT has little effect on the performance. A short conclusion from these observations is that a maximum SRT between 0.1 and 0.2 is preferred, and the minimum SRT can be any value greater than 0.5.

Table 4.8 The performance with different SRT settings.

Test #	*Min SRT*	*Max SRT*	Detection accuracies on all samples	Classification accuracies on faulty samples
14	*0.6*	*0.0*	71.4%	35%
15	*0.6*	*0.05*	69.6%	50%
16	*0.6*	*0.1*	76.8%	55%
17	*0.6*	*0.15*	76.8%	50%
18	*0.6*	*0.2*	71.4%	60%
19	*0.6*	*0.25*	73.2%	55%
20	*1.0*	*0.0*	67.9%	25%
21	*1.0*	*0.05*	69.6%	30%
22	*1.0*	*0.1*	76.8%	45%
23	*1.0*	*0.15*	75.0%	60%
24	*1.0*	*0.2*	75.0%	60%
25	*1.0*	*0.25*	75.0%	60%

4.6.4 The Probability Threshold

The probability threshold is used to determine whether the output of the classifier should be treated as a fault. For example, if the threshold is set to 0.9 and the classifier shows that fault 1 has the highest probable rating of 0.85, then the sample would be considered as normal and free of fault. Given a trained model, a larger probability threshold generally results in more samples being classified as normal, leading to a higher True Negative rate (TN rate, the accuracy on normal samples). However, the True Positive rate (TP rate) generally reduces in the meantime. Setting a smaller threshold has the opposite effect. Therefore, configuring the probability threshold balances the performance metrics and is often based on practical concerns. Changing the threshold does not alter the characteristics of a model, but it provides a comprehensive view of the performance.

In the previous subsections, the accuracy results are obtained by manually setting best-performing probability thresholds, whose values are absent in the tables to avoid confusion. This section takes the tests #14-25 in Table 4.7, Section 4.6.3 and examines the threshold's effect in detail. When the threshold changes, the two accuracy values – the TP rate and the TN rate – also change accordingly. A set of TP and TN values is obtained as the threshold sweeps from 0.0 to 1.0. Plotting the two variables on a 2D graph gives the True-Positive-True-Negative (TPTN) curve. It is an adaptation of the more commonly known Receiver Operating Characteristic (ROC) curve, which plots TP rate vs False Negative (FN) rate. The TPTN curve is chosen for presentation because these two metrics are more intuitive in this study.

Figure 4.6.1 shows the TPTN curve from tests #14-25. The figure on the left and right present the tests with maximum SRT=0.6 and 1.0, respectively. Each circle on the curve represents a threshold value. Its x and y coordinates are the TP rate and TN rate. The curves start at (1.0, 0.0), where the threshold is 0.0. At this point all samples are classified as faulty, so the TP rate is 1.0 and the TN rate is 0.0. As the threshold increases, the curve traverse through the unit square towards (0.0, 1.0), where the threshold is 1.0. Higher values for both metrics are preferred, so the curves that bend towards the upper right corner (1.0, 1.0) are considered as having good character. To measure this character, the Area Under Curve (AUC) metric is often used. In this demonstration, AUC is observed instead of precisely calculated.

For both graphs, the minimum SRT values in the range of [0.1, 0.25] perform better than the values in the range of [0.0, 0.05] in terms of AUC. This coincides with the observation from Table 4.8. Comparing the two graphs side by side, the curves in the right graph (maximum SRT=1.0) show slightly larger AUC than the ones on the left. This is a new finding that can hardly be obtained from examining accuracy tables. Only relying on AUC, the best-performing configurations are minimum SRT=0.15 and 0.25 with maximum SRT=1.0. However, since the curve with a larger AUC does not fully cover the smaller ones, it is better to consider multiple curves when picking a value for a certain application.

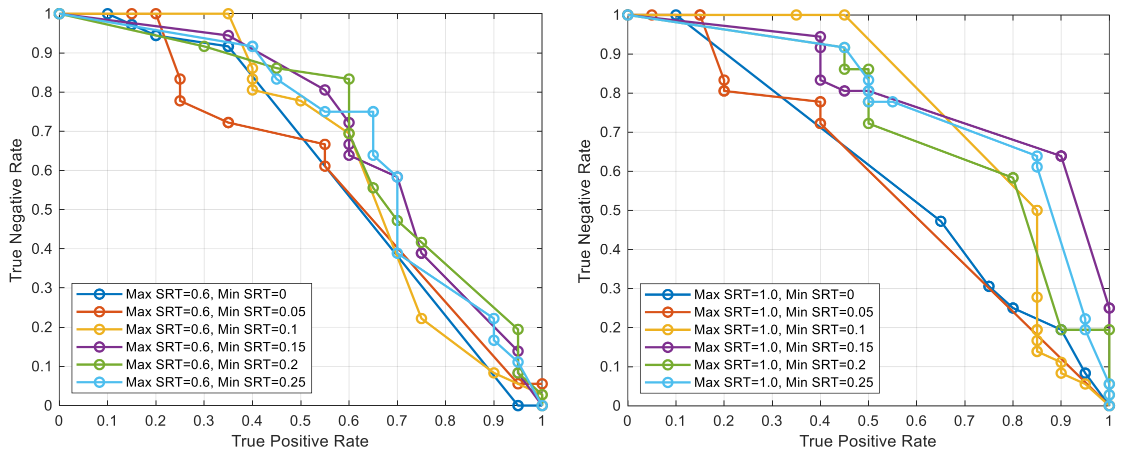


Figure 4.6.1 The effect of probability thresholds.

In general, the points closer to (1.0, 1.0) on the TPTN graph are considered as good configurations that have a balanced performance. If certain metrics are strictly required, the graphs can help quickly determine the best setting. For example, if a TN rate of 80% is required, the highest possible TP rate can be found by using Figure 4.6.1. On the left figure, the eligible configuration is at (0.6, 0.84) on the curve of maximum SRT=0.6, minimum SRT=0.2, the green plot. On the right figure, the eligible configuration is at (0.5, 0.86) on the curve of maximum SRT=1.0, minimum SRT=0.2, also the green plot. Clearly the former configuration is better as it provides 10% higher TP rate while satisfying the TN rate requirement. Therefore, multiple curves and graphs should be considered when choosing the best model configurations.

4.6.5 Discussion

This section examines the effects of gap constraints, maximum lengths, and SRTs in the pattern extraction process, as well as the probability thresholds in the detection process. An empirical tuning approach is developed, and the best-performing configurations in terms of AUC are maximum pattern length=7, gap=0, maximum SRT=1.0, minimum SRT=0.15 or 0.25. This best-performing configuration might vary case by case in practice, but the tuning approach can be adopted universally.

It is worth noting that the absolute values presented in this section are dependent on the dataset. There is a serious lack of training data issue in this study, especially the number of samples in each label. Given the fact that a majority of the faulty labels contain only two training samples, an overall >70% (greater than 70%) detection rate and >50% (greater than 50%) classification rate is more than sufficient to demonstrate the effectiveness of the methodology. There is a large potential for improvement if more training data are available.

In terms of computation, despite the ConSGapMiner mining algorithm showing great efficiency when the configuration values change, the training process is still time-consuming. The pattern discovery is likely to cost more time and memory if more training data and faults are included. The detection phase is relatively lightweight and costs little time when testing on the performance PC. However, if deploying on a mobile platform or perform online detection, the pattern-matching operation during the detection phase may potentially pose a significant computational overhead.

4.7 Conclusion and Discussion

This chapter presents the FDSPM diagnosis model. FDSPM is the first to incorporate contrast mining algorithms – in this case, the ConSGapMiner – to extract fault patterns from log token samples. The number of patterns can be large depending on the mining configuration, presenting the multi-feature, multi-class classification problem. FDSPM uniquely solves the problem by rating the features through knowledge base and applying a naïve Bayes classifier. The FDSPM has demonstrated acceptable detection and classification performance, using a minimal amount of training data.

Numerous parameters need to be properly configured in order to achieve better performance. The effect of each individual parameter is closely examined in a series of comparison tests. A comprehensive tuning method is developed and gradual improvement is observed over the tuning process. The obtained parameters can be case-specific, meaning that when additional data are available for training, re-tuning using the same method is required.

One practical concern of the FDSPM is the computation requirement. Although the mining algorithm has demonstrated good efficiency, learning each fault can still take minutes. This makes the already cumbersome tuning process even more time-consuming. The pattern-matching operation during the detection phase can also pose computational issues on mobile platforms with limited processing power.

The next chapter will examine a different machine learning approach using statistical features instead of sequence patterns, which would address the efficiency issue and simplify the tuning process.

Chapter 5 Statistical Features and Machine Learning

The previous chapter presents a fault diagnosis system through mining sequence patterns within log data. Although intuitive, it requires a cumbersome sequence-based pattern searching process. This chapter introduces a different diagnosis approach using statistical features and numerical machine learning models. The framework is called Fault Diagnosis with Statistical Machine Learning (FDSML). The FDSML framework incorporates a process called vectorization, which converts the log sequences from the token format into numerical feature vectors. A machine learning model then processes the numerical features and produces a classification of possible faults. Both the vectorization and machine learning process have plenty of methods to choose from, making the FDSML a flexible framework for diagnosis.

The feature vectorization process emphasises the statistical features of the log sequences. A number of such techniques are demonstrated in existing studies including [51]–[59], [61]–[63] reviewed previously in Section 3.4. The vectorization method used in this study originates from textual analysis and is elaborated in Section 5.2.2.

Machine learning as a general term refers to a computer program whose performance at certain tasks improves through training experience [10], such as the processing of a dataset. Based on this definition, all methods presented in this thesis are under the broad category of machine learning. However, conventionally when referring to machine learning, people think of it as the narrow category of using numerical models to approximate a function that describes a given dataset. Moreover, the recently developed deep neural networks (also referred to as deep learning), which originates from and shares many common methodologies with the narrow term of machine learning, are generally regarded as a different topic. The sections in this chapter follow this convention when using the term machine learning wherever applicable. The machine learning models for the FDSML framework are introduced in Section 5.2.3.

5.1 Why Moving to Numerical?

As concluded in Chapter 4, one of the drawbacks of using sequential patterns and naïve Bayes classifier is the long execution time. Pattern searching is shown to be a computationally heavy process even with efficient data mining algorithms. The log sequences being categorical sequences in nature render many efficient numerical methods inapplicable. If the log sequences are converted into numerical representations, there would be an abundance of candidate machine learning models available to apply or extend upon. In other words, the representative features for software faults can be numerical and they are not necessarily constrained to a sequential format.

Vectorization is the process that converts categorical tokens into numerical vectors. Most of these techniques require little computational resources. Many methods mentioned in the literature [56], [57], including the count and frequency of tokens, can be obtained with one scan of the sequence, i.e., the process has a constant-time complexity $\mathcal{O}(1)$. This is unsurprising because these techniques are developed for big data applications with an emphasis on efficiency. For example, some text processing applications successfully applied vectorization techniques to the whole English Wikipedia dataset, which comprises over two million samples [84]. Traditional data mining approaches like the one introduced in Chapter 4 would be infeasible for such a large amount of data.

Machine learning algorithms are often very efficient as well. The forward propagation of a machine learning model, i.e., taking a data sample as input and producing an output, is essentially a one-time computation of a numerical function. The training process of such models is a numerical optimisation problem where many solutions apply. On the other hand, the models that work directly with sequence data, such as the Finite State Machine (FSM) and Bayesian network, are far from efficient. The improvement in computation efficiency can also save time for the model tuning process.

The FDSML framework consists of two phases: the training phase and the diagnosis phase, as shown in Figure 5.1.1. The log sequence goes through the tokenization and vectorization process to obtain their feature vectors. These vectors are used to train a numerical machine learning model which, after training, is deployed to detect and diagnose system faults from log data. The methods in each process are elaborated in Section 5.2 and the implementation is explained in more detail in Section 5.3.

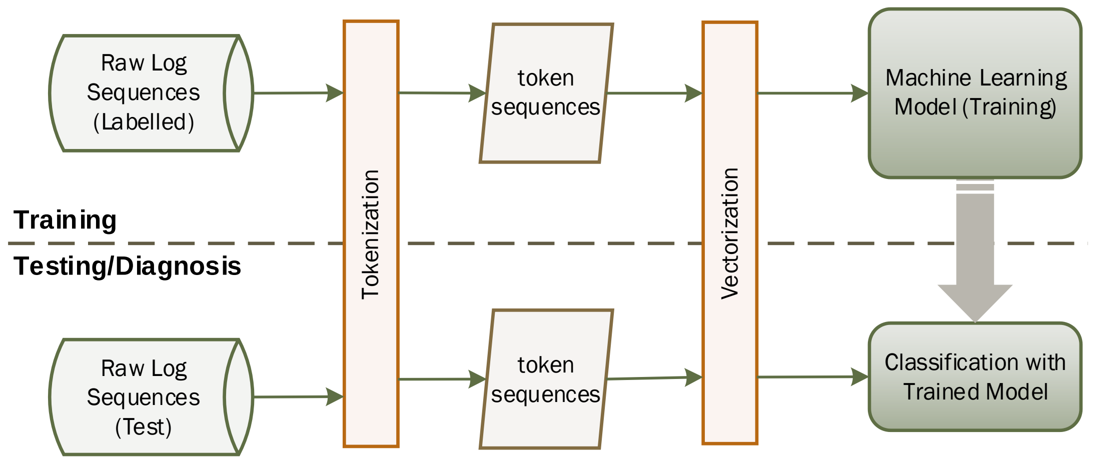


Figure 5.1.1 The overview of fault diagnosis with vectorization and machine learning.

5.2 Methodology

The three main processes of the FDSML framework are tokenization, feature vectorization, and classification using a machine learning model. They are introduced in the following subsections.

5.2.1 Tokenization

The log samples are first converted into token sequences, a process called tokenization, the same as the one introduced in the previous chapter. The tokenization method from Section 4.2.2 is used in this research. After tokenization, the log sample is represented by a sequence of tokens represented by numbers.

5.2.2 Vectorization

The term vectorization means, literally, converting a data sample into a vector. This vector is also called a *feature vector*, representing some abstract features of the data sample. By convention, a feature vector is a column vector with a size of m. Each feature value – the value of each row – has a definitive meaning. These definitions are often pre-defined and remain consistent across all data such that different samples' feature vectors are comparable. The following subsections introduce a few generic vectorization methods for sequence data. These vectorization processes can be viewed as using manually specified rules to extract features from data and therefore, they are also called feature engineering.

5.2.2.1 Term Count

Term count is a simple technique that counts the occurrences of each unique token within a sample sequence. The unique token is commonly referred to as a *term* in information retrieval literature. The terminologies *token* and *term* are used interchangeably in this section. In a term count vector, each row value represents the count of a term. Take the sequence "ABCDA" as an example in Figure 5.2.1. Define each row as the count of terms A, B, C, and D, the term count vector is then $\boldsymbol{v}_{\mathrm{tc}} = [2, 1, 1, 1]'$. The term A occurs twice and all the rest terms occur once.

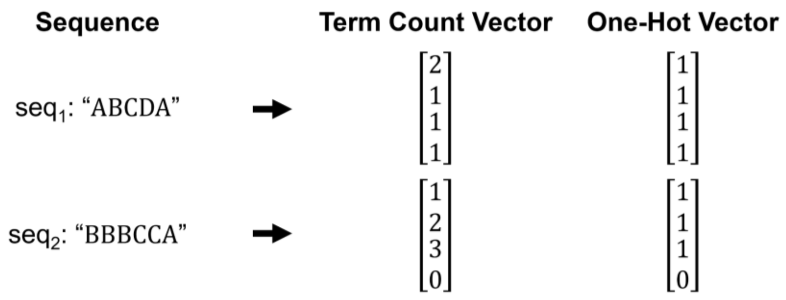

Sequence
Term Count Vector
One-Hot Vector
seq₁: "ABCDA"
seq₂: "BBBCCA"
$\begin{bmatrix}2\\1\\1\\1\end{bmatrix}$ $\begin{bmatrix}1\\1\\1\\1\end{bmatrix}$
$\begin{bmatrix}1\\2\\3\\0\end{bmatrix}$ $\begin{bmatrix}1\\1\\1\\0\end{bmatrix}$

Figure 5.2.1 Example of the term count vector.

The order of terms in a term count vector must be pre-defined and remain unchanged throughout the process. This is necessary to ensure the conversion is consistent across all sequence samples. If a term does not occur at all, it would have a count of 0 instead of being omitted. Applying the same conversion to the sequence “BBCCCA”, the term counter vector is then $[1, 2, 3, 0]'$. Note that regardless of the length of the sequence, vectorization always produces an output with the same length equal to the total number of unique tokens.

One-hot vectorization is another common method similar to the term count. For the terms with non-zero counts, the one-hot vector simply assigns a value of 1 instead of counting the occurrence. The resulting vector consists of either 1 or 0 values, as shown in Figure 5.2.1. The term one-hot originates from digital circuits where the bit values are either ‘1’ (hot) or ‘0’ (cold). One-hot vector is effective to indicate which tokens are present in the sequence and in the meantime, avoids issues that may be caused by large term counts. However, in machine learning research where floating points are widely used, restricting the values to binary tend to limit the potential of feature extraction.

5.2.2.2 Term Frequency

The absolute values of a term count vector may be less meaningful when a large number of occurrences exist, especially when the sequence samples have different lengths. For example, if a sequence sample S_1 is 10 times longer than another sample S_2, the term counts of S_1 are likely a lot larger than that of S_2. As a result, the same values of a term in S_1 and S_2 would mean differently, because the sample length is different. Term Frequency (TF) is defined as the count divided by the total number of tokens in a sequence sample. It is therefore a better representation than the term count in many cases. Given the term count vector $\boldsymbol{v}_{\mathrm{tc}}$, the TF vector $\boldsymbol{v}_{\mathrm{tf}}$ can be represented as:

$$\boldsymbol{v}_{\mathrm{tf}} = \frac{\boldsymbol{v}_{\mathrm{tc}}}{\sum_{i=0}^{n} \boldsymbol{v}_{\mathrm{tc},i}} \tag{5.1}$$

The TF vectors of the two previous sequence examples are shown Figure 5.2.2. The TF vector scales down a term count vector to a range of [0, 1]. This gives a better statistical description of the data. Since the large values are absent, the TF feature is also preferable by machine learning models, as larger values may cost the training more time to reach convergence with a gradient based algorithm that is explained later in this section.

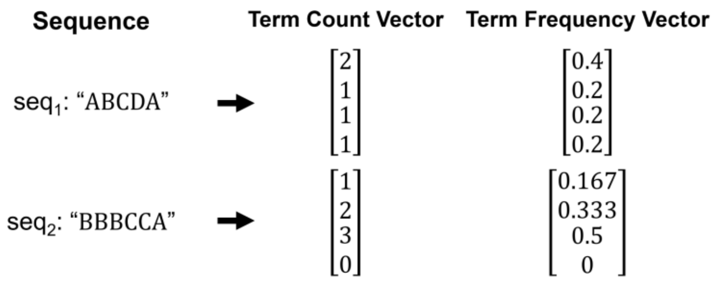

Sequence
Term Count Vector
Term Frequency Vector
seq$_1$: "ABCDA"
$\begin{bmatrix}2\\1\\1\\1\end{bmatrix}$
$\begin{bmatrix}0.4\\0.2\\0.2\\0.2\end{bmatrix}$
seq$_2$: "BBBCCA"
$\begin{bmatrix}1\\2\\3\\0\end{bmatrix}$
$\begin{bmatrix}0.167\\0.333\\0.5\\0\end{bmatrix}$

Figure 5.2.2 Example of the term frequency vector.

Although the term count and TF vectors are relatively simple, they are effective techniques. The combination or adaptation of both can be found in many log data analysis studies [56]–[58].

5.2.2.3 Inverse Document Frequency

If treating the feature vector as an importance rating of the tokens, the TF vector implies that all terms are *inherently* equally important. As a result, more attention is drawn to the terms with a higher TF value, which is directly linked to more occurrences. However, a higher TF value does not necessarily mean higher importance. It is very likely that tokens occurring more in a sequence represent a less meaningful feature. For example, a frequently executed task under normal conditions can leave a large number of duplicated log traces, creating large values in the TF vector. These large values are less desired in a fault diagnosis setting because they represent a normal execution and benign log tokens. The actual fault patterns tend to appear less frequently but more distinctive among faulty samples.

To address this issue, terms that occur too often across the dataset need to be attenuated from the TF vector. The document frequency (DF) feature is defined as the number of samples in the dataset that contains a term (token) t. The DF value of a term t is then:

$$\mathrm{df}_t = \frac{\text{number of samples containing } t}{\text{number of total samples}} \tag{5.2}$$

Using the same example in the previous subsection, assuming the dataset contains only two sequence samples shown in Figure 5.2.3. The term A, B, and C appears in both samples, so they have a DF value of 1.0. The term D only occurs in the first sequence, so it has a DF value of 0.5. the DF vector of all terms is also shown in Figure 5.2.3. Note that only one DF vector is produced for each dataset, instead of each sample.

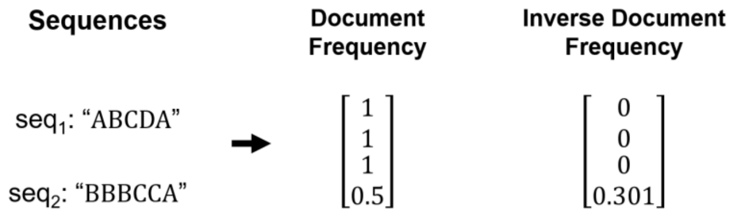

Sequences
Document Frequency
Inverse Document Frequency
seq$_1$: "ABCDA"
seq$_2$: "BBBCCA"
$\begin{bmatrix}1\\1\\1\\0.5\end{bmatrix}$
$\begin{bmatrix}0\\0\\0\\0.301\end{bmatrix}$

Figure 5.2.3 Example of the inverse document frequency vector.

In a DF vector, the larger value represents higher occurrences in the dataset. The opposite is more useful. In other words, the terms that occur rarely should have a higher weighting. Inverse Document Frequency (IDF) is introduced to flip the polarity. A common definition also includes logarithms to scale down the unexpected large values:

$$\mathrm{idf}_t = \log \frac{1}{\mathrm{df}_t} \tag{5.3}$$

The IDF feature in a vector format is expressed as:

$$\boldsymbol{v}_{\mathrm{idf}} = \begin{bmatrix} \mathrm{idf}_1 \\ \mathrm{idf}_2 \\ \mathrm{idf}_3 \\ \dots \end{bmatrix} \tag{5.4}$$

The IDF vectors in the previous example are also shown in Figure 5.2.3.

5.2.2.4 TF-IDF Feature

The term-frequency-inverse-document-frequency (TF-IDF) vector combines the best of two previous features. Specifically, for each sample, the TF-IDF vector is obtained by element-wise product of the sample's TF vector and the dataset's IDF vector:

$$\boldsymbol{v}_{\mathrm{tf-idf}}^{(i)} = \boldsymbol{v}_{\mathrm{tf}}^{(i)} \odot \boldsymbol{v}_{\mathrm{idf}} \tag{5.5}$$

where $\odot$ denotes element-wise multiplication, or Hadamard product. The superscript (i) denotes the i-th samples in the dataset.

Using the same examples as before, the TF-IDF vectors of two sequences are shown in Figure 5.2.4. Their TF and IDF vectors are copied from the previous two figures. The TF-IDF vector's values capture a term's frequency in a sequence as well as its scarcity among all sequence samples. Specifically, the term "D" only appears in sequence 1, so it has a value of 0.06 in the first TF-IDF vector, while its value is zero for the second TF-IDF vector.

Applying TF-IDF vectorization to the log sequences, each sample is converted into a feature vector $\boldsymbol{x}^{(i)}$ with a fixed length m that equals the total number of unique tokens.

This vector is used as the input feature for the machine learning process introduced in the following section.

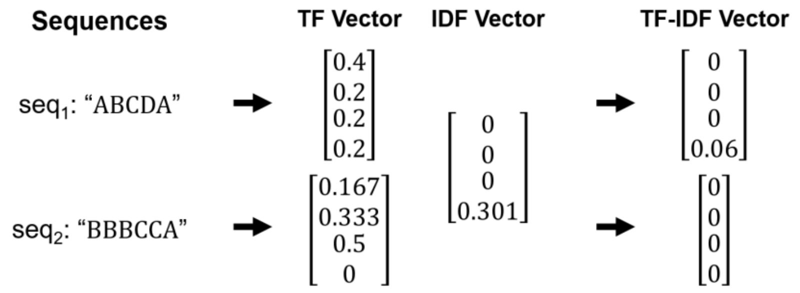


Figure 5.2.4 Example of the TF-IDF vector.

5.2.3 The Machine Learning Classification Model

The machine learning classification model is the final step of the automated diagnosis framework. The input of the model is the feature vectors and the output is a classification category, i.e., the possible faults within the log sample. As a machine learning model, a training process is required. Training means the model adjusts its parameters to fit available data. After the model is trained, it is expected to produce reasonable classifications or predictions.

Machine learning algorithms are divided into supervised learning and unsupervised learning, depending on whether class labels are required. In this study, the class labels are the predefined faults as shown in the previous chapter. Unsupervised learning does not require such labels, eliminating the need for manual labelling. The unsupervised approach is therefore much attractive to the applications where the dataset is too large to manually label. However, unsupervised learning can produce unpredictable results when applying to multi-class problems. For example, the k-means clustering algorithm, a popular unsupervised learning method, requires a pre-defined number of class labels and isotropic distributed data to generate meaningful results, otherwise it may generate clusters that achieve theoretical optimality but are practically unusable [85]. Therefore, unsupervised learning more commonly applies to anomaly detection problems through finding isolated outliers rather than diagnosis problems.

Supervised learning methods utilize the label information during the training process. Each training sample has a label indicating the ground truth, which in this study is the actual fault. The labels act as guidance during the training process, as the supervised learning model adjusts its parameters to get the output as close to the ground truth as possible. As a result, a supervised learning method is almost always better than an unsupervised learning method, especially in a multiclass classification scenario. This section introduces some common supervised learning methods and applies them to the fault diagnosis problem.

5.2.3.1 Linear Regression

Linear regression attempts to fit a linear equation to the training data $\boldsymbol{x}$ such that the outcome approximates the ground truth $\boldsymbol{y}$. The ground truth $\boldsymbol{y}$ is a continuous numerical value instead of a categorical one, so the linear regression is called a prediction model. It therefore does not apply directly to the fault classification problem. However, as the fundamental component of logistic regression and artificial neural networks, the detail of linear regression is worth to be elaborated here.

Denote the whole training data containing m samples as $\boldsymbol{x} = [\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(3)}, \dots, \boldsymbol{x}^{(m)}]$. Each sample $\boldsymbol{x}^{(i)}$ is a column vector, such as the $\boldsymbol{v}_{\text{tf-idf}}$ vector in Section 5.2.2.4. Denote the ground truth values as $\boldsymbol{y} = [y^{(1)}, y^{(2)}, y^{(3)}, \dots, y^{(m)}]$. For the i-th sample, $\boldsymbol{x}^{(i)} = \left[x_1^{(i)}, x_2^{(i)}, x_3^{(i)}, \dots\right]^T$ are the feature values. A linear regression fits a set of parameters $\boldsymbol{w} = [w_0, w_1, w_2, \dots]^T$ using a linear equation $h(\boldsymbol{x}^{(i)}, \boldsymbol{w})$:

$$h\left(\boldsymbol{x}^{(i)}, \boldsymbol{w}\right) = w_0 + w_1 x_1^{(i)} + w_2 x_2^{(i)} + \cdots \tag{5.6}$$

$$h\left(\boldsymbol{x}^{(i)}, \boldsymbol{w}\right) = \boldsymbol{w}^T \boldsymbol{x}^{(i)} \tag{5.7}$$

Equation (5.7) is the matrix representation of (5.6). Note that strictly, because of the term w_0 in (5.6), the right side of equation (5.7) should be $\boldsymbol{w}^T[1, \boldsymbol{x}^{(i)}]$. Here for simplicity, the term $\boldsymbol{x}^{(i)}$ represents the modified sample data $[1, \boldsymbol{x}^{(i)}]$ for the rest of this chapter where applicable. In the literature, a different way of writing equation (5.7) is $h\left(\boldsymbol{x}^{(i)}, \boldsymbol{w}, b\right) = \boldsymbol{w}^T \boldsymbol{x}^{(i)} + b$, which uses a separate term b called *bias* to replace w_0 in order to differentiate itself from the weight vector $\boldsymbol{w}$. The mathematics remains the same in both representations.

In machine learning, the parameters $\boldsymbol{w}$ are also called weights, the model's output $h(\boldsymbol{x}^{(i)}, \boldsymbol{w})$ is commonly denoted as $\hat{y}^{(i)}$. A graphical representation of the linear regression model is shown in Figure 5.2.5 using a sample $\boldsymbol{x}^{(i)}$ with three features.

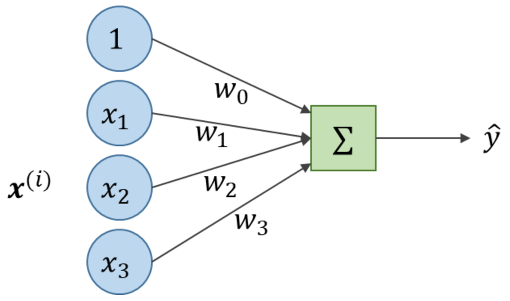


Figure 5.2.5 Example of the linear regression model.

The approximation of $\hat{y}^{(i)}$ with respect to $y^{(i)}$ is measured by a cost function $J(\boldsymbol{w})$. A commonly used measurement is the Mean Square Error (MSE) between the model output and the ground truth for all m samples:

$$J(\boldsymbol{w}) = \frac{1}{m}\sum_{i=1}^{m}\frac{1}{2}\left[\hat{y}^{(i)} - y^{(i)}\right]^2 \tag{5.8}$$

where the $1/2$ term within the summation is to facilitate partial derivative calculation.

A common method to find the optimal $\boldsymbol{w}$ is *gradient descent*. It starts with a random $\boldsymbol{w}$, which produces a non-optimal $J(\boldsymbol{w})$. The algorithm incrementally moves the current $\boldsymbol{w}$ towards the opposite direction of the gradient $\nabla J(\boldsymbol{w})$. Since the gradient is the direction and rate of fastest increase, the method gets its name as gradient descent. The update of $\boldsymbol{w}$ is done iteratively with a small step α, referred to as the learning rate. The gradient descent algorithm can be expressed as:

$$\boldsymbol{w}_{(k+1)} \coloneqq \boldsymbol{w}_{(k)} - \alpha\nabla J_{(k)}(\boldsymbol{w}) \tag{5.9}$$

The operator $\coloneqq$ is colon equals, meaning the term on the left is defined as the term on the right. The subscript in a bracket (k) indicates the iteration step.

Explicitly, when the sample $\boldsymbol{x}^{(i)}$ is being trained, the gradient at point $\boldsymbol{w}$ can be calculated as follows based on equations (5.7) and (5.8):

$$\nabla J(\boldsymbol{w}) = \frac{\partial J(\boldsymbol{w})}{\partial \boldsymbol{w}} = \frac{1}{m}\sum_{i=1}^{m}\left[\hat{y}^{(i)} - y^{(i)}\right] \cdot \boldsymbol{x}^{(i)} \tag{5.10}$$

Note that this equation specifies the gradient for one sample. One update of $\boldsymbol{w}$ is made when processing each $\boldsymbol{x}^{(i)}$ according to this equation. This implementation is called Stochastic Gradient Descent (SGD). Correspondingly, there is batch gradient descent. Batch gradient descent performs one update of $\boldsymbol{w}$ when processing the whole training dataset $\boldsymbol{x}$. The gradient $\nabla J(\boldsymbol{w})$ in (5.10) is the average of all samples' gradients. Batch gradient produces smoother and possibly faster convergence than SGD, but larger datasets may lead to memory shortage because every calculation uses the whole dataset. Mini-batch gradient descent is the compromise between the two, using a certain number (mini-batch) of samples to calculate $\nabla J(\boldsymbol{w})$ and update $\boldsymbol{w}$. Mini-batch and stochastic gradient descent are two of the most commonly used optimization algorithms for machine learning applications.

5.2.3.2 Logistic Regression

For classification problems, the training data $\boldsymbol{x} = [\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(3)}, \dots, \boldsymbol{x}^{(m)}]$ have the same format as regression problems, but the labels $\boldsymbol{y}$ contain only categorical values that are called classes. For example, the binary classification requires the class to be either 0 or 1. One way to adapt the linear regression to fit the classification problem is to scale the output value to the range of (0,1), and then apply thresholding to obtain a class label of either 0 or 1. The logistic function is a common way of mapping continuous values to a certain range. Specifically, the sigmoid function, a form of the logistic function, is defined as follows:

$$S(z) = \frac{1}{1 + e^{-z}} \tag{5.11}$$

The sigmoid function has a characteristic S-shaped curve and $S(z)$ has an upper limit of 1 and a lower limit of 0. Applying the sigmoid function to the linear regression model's output, the logistic regression model is defined as follows:

$$\begin{gathered} z = \boldsymbol{w}^T \boldsymbol{x}^{(i)} \\ \hat{y} = S(z) = \frac{1}{1 + e^{-z}} \end{gathered} \tag{5.12}$$

The output $\hat{y}$ is within the range of (0,1). It can be interpreted as the probability of a class label of 1 for the given sample $\boldsymbol{x}^{(i)}$. A graphical representation of the logistic regression is shown in Figure 5.2.6.

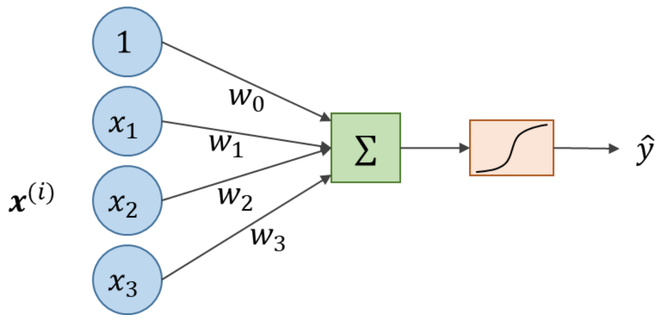


Figure 5.2.6 Example of the logistic regression model.

A cost function is defined for logistic regression as well. Because the error term $\hat{y}^{(i)} - y^{(i)}$ is always less than 1, directly using the MSE becomes less effective. Therefore, the logarithm is used in the cost function. Specifically:

$$J^{(i)}(\boldsymbol{w}) = \begin{cases} -\log\left(\hat{y}^{(i)}\right), & y^{(i)} = 1 \\ -\log\left(1-\hat{y}^{(i)}\right), & y^{(i)} = 0 \end{cases} \tag{5.13}$$

In this way, the cost function amplifies the error term and is always non-negative. This loss function is called *categorical cross-entropy loss* or *log loss*. Note that in this equation only one sample is calculated, as indicated by the superscript (i). A compact form of writing the cross-entropy loss is:

$$J^{(i)}(\boldsymbol{w}) = -y^{(i)}\log\hat{y}^{(i)} - \left(1-y^{(i)}\right)\log\left(1-\hat{y}^{(i)}\right) \tag{5.14}$$

The average log loss for the whole dataset is:

$$J(\boldsymbol{w}) = \frac{1}{m}\sum_{i=1}^{m}\left[-y^{(i)}\log\hat{y}^{(i)} - \left(1-y^{(i)}\right)\log\left(1-\hat{y}^{(i)}\right)\right] \tag{5.15}$$

The gradient descent algorithm in (5.9) also applies to logistic regression. The gradient term can be obtained using the derivative chain rule:

$$\nabla J(\boldsymbol{w}) = \frac{\partial J(\boldsymbol{w})}{\partial \boldsymbol{w}} = \frac{\partial J(\boldsymbol{w})}{\partial \hat{\boldsymbol{y}}}\frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{z}}\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{w}} \tag{5.16}$$

5.2.3.3 Multiclass Logistic Regression

The logistic regression algorithm presented in the previous chapter applies to binary classification only. In a multiclass scenario with K different class labels, a "one-vs-all" approach can be adopted. Each class label is treated individually and processed by a separate logistic regression model. Concretely, for the sample $\boldsymbol{x}^{(i)}$:

$$\hat{y}_1^{(i)} = \frac{1}{1 + e^{-\boldsymbol{w}_1^T \boldsymbol{x}^{(i)}}}$$
$$\hat{y}_2^{(i)} = \frac{1}{1 + e^{-\boldsymbol{w}_2^T \boldsymbol{x}^{(i)}}} \tag{5.17}$$
$$\dots$$
$$\hat{y}_K^{(i)} = \frac{1}{1 + e^{-\boldsymbol{w}_K^T \boldsymbol{x}^{(i)}}}$$

Therefore, the output of the "one-vs-all" model is now a vector $\hat{\boldsymbol{y}}^{(i)} = \left[\hat{y}_1^{(i)}, \hat{y}_2^{(i)}, \dots, \hat{y}_K^{(i)}\right]^T$ instead of a single value. Each value in the vector $\hat{\boldsymbol{y}}^{(i)}$ represents the probability of a class. The weights for the whole model $\boldsymbol{w} = [\boldsymbol{w}_1, \boldsymbol{w}_2, \dots, \boldsymbol{w}_K]$ are now a 2D matrix. Writing in the matrix format, the formula for multiclass logistic regression is:

$$\hat{\boldsymbol{y}}^{(i)} = \frac{1}{1 + e^{-\boldsymbol{w}^T \boldsymbol{x}^{(i)}}} \tag{5.18}$$

Correspondingly, the class labels $\boldsymbol{y}$ are converted to one-hot vectors (similar to the one-hot vector in Section 5.2.2 Vectorization) to match the format of the model's output $\hat{\boldsymbol{y}}$. An example of the conversion is shown in Figure 5.2.7. In this example, a label $y^{(i)} = 3$ is represented by the vector $[0, 0, 1, 0]^T$.

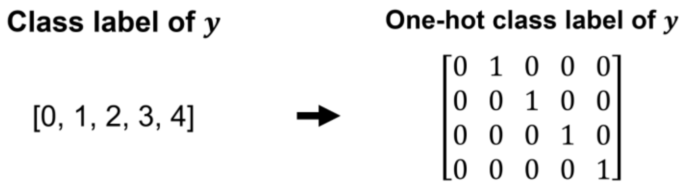

Class label of y
One-hot class label of y
[0, 1, 2, 3, 4]
$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$

Figure 5.2.7 Converting class label into one-hot vector.

Now the logistic regression model's multiclass outputs match the one-hot class labels. However, each probable value in the output vector $\hat{\boldsymbol{y}}^{(i)}$ only accounts for the class itself.

As a result, there could be multiple classes all having high probable scores higher than 0.5, which defies the purpose of probable scores. The output vector needs to be normalized such that each value properly represents the probability of an output class. This is achieved through a normalized exponential function, or softmax function:

$$\sigma(\boldsymbol{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{5.19}$$

where $\boldsymbol{z} = [z_1, z_2, \dots z_K]$ is the input vector with length K and $\boldsymbol{\sigma} = [\sigma(\boldsymbol{z})_1, \sigma(\boldsymbol{z})_2, \dots, \sigma(\boldsymbol{z})_K]$ is the normalized vector.

The multiclass logistic regression model with softmax normalization is represented as follows. It is also called multinomial logistic regression.

$$\begin{gathered} \boldsymbol{z} = \boldsymbol{w}^T \boldsymbol{x}^{(i)} \\ \hat{\boldsymbol{y}}^{(i)} = \text{softmax}(\boldsymbol{z}) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \end{gathered} \tag{5.20}$$

A graphical representation of this model is shown in Figure 5.2.8.

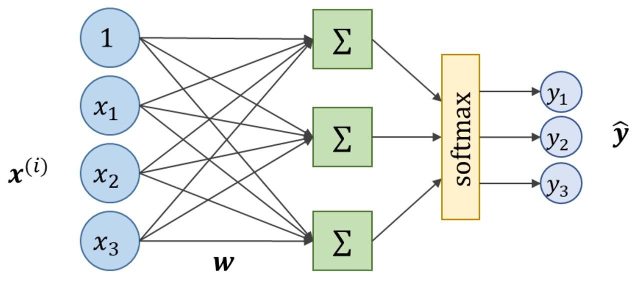


Figure 5.2.8 Example of the multiclass logistic regression model.

The loss function for the multiclass setting is also the categorical cross-entropy. For the sample $\boldsymbol{x}^{(i)}$ and the model output $\hat{\boldsymbol{y}}^{(i)}$ and the ground truth label in one-hot form $\boldsymbol{y}^{(i)}$, the loss is defined as:

$$J(\boldsymbol{w})^{(i)} = -\sum_{k=1}^{K} y_k^{(i)} \log \hat{y}_k^{(i)} \tag{5.21}$$

For the whole dataset, the loss function in a batch form is:

$$J(\boldsymbol{w}) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log \hat{y}_k^{(i)} \tag{5.22}$$

Note that these equations also apply to the special case of binary classification represented by equation (5.14). The gradient can be obtained using the derivative chain rule as well.

5.2.3.4 Artificial Neural Network

Both linear and logistic regression models are curve fitting techniques that use a function $\hat{y} = f(\boldsymbol{w}, \boldsymbol{x})$ to approximate the actual process that describes the data points. The actual process can be very complex that the model's weights are not sufficient to produce a near-optimal function. In such cases, a more complex model is required.

Each binary logistic regression model can be viewed as a unit as shown in Figure 5.2.9, taking an input vector of a pre-defined size and producing an output. A large number of these units can interconnect to create a network to approximate more complex functions. The result is a structure called an Artificial Neural Network (ANN), and each unit is a *neuron*. The logistic function is called the *activation* of a neuron. This naming convention was meant to simulate the functioning of the human brain. However, more recent development of ANN has generally deviated away from biological imitation. For example, many models adopt simpler activations other than logistic functions to reduce computational complexity.

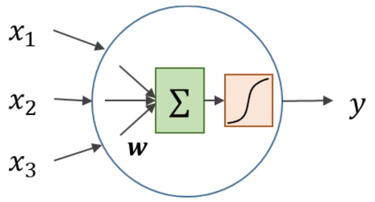


Figure 5.2.9 Illustration of a single neuron.

Multi-Layer Perceptron (MLP) is a common way of organising neurons by layers as shown in Figure 5.2.10. Each layer contains a certain number of neurons, whose input and output are connected to the previous layer's output and the next layer's input, respectively. Each neuron is connected to every neuron in the neighbouring layer, so an MLP is also called a fully connected network. The layers that are not adjacent have no connections at all. The output layer contains the same amount of neurons as the number of class labels (given a classification setting). A softmax activation is used in the same way as in the multiclass logistic regression model. In an MLP, the number of layers and the number of neurons in each layer are configurable to achieve a better fit for the data. Therefore, such a model is highly versatile for complex problems.

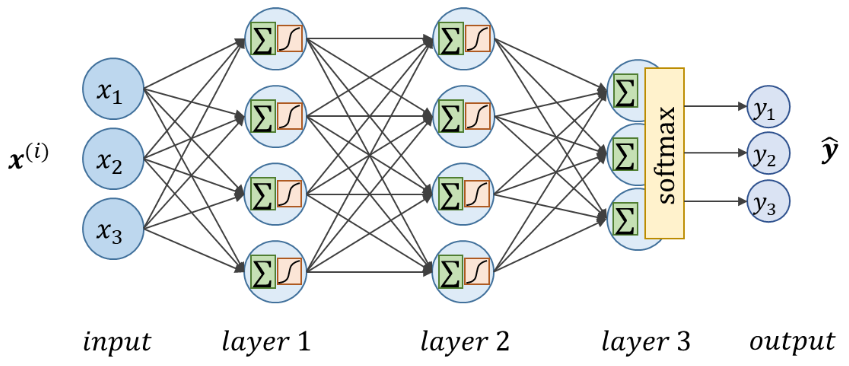


Figure 5.2.10 Example of a Multi-Layer Perceptron (MLP) model.

The mathematical representation for a d-layer MLP can be written layer by layer as follows:

$$
\begin{aligned}
\boldsymbol{z}_1 &= S\left(\boldsymbol{w}_1^T \boldsymbol{x}^{(i)}\right) \\
\boldsymbol{z}_2 &= S(\boldsymbol{w}_2^T \boldsymbol{z}_1) \\
\boldsymbol{z}_3 &= S(\boldsymbol{w}_3^T \boldsymbol{z}_2) \qquad (5.23) \\
&\dots \\
\hat{\boldsymbol{y}}^{(i)} &= \boldsymbol{z}_d = \text{softmax}(\boldsymbol{w}_d^T \boldsymbol{z}_{d-1})
\end{aligned}
$$

where $\boldsymbol{z}_1$ through $\boldsymbol{z}_d$ are the output of each corresponding layer's neurons in a vector form, $\boldsymbol{w}_1$ through $\boldsymbol{w}_d$ are the weight matrices of each corresponding layer's neurons, S is the choice of activation function for that layer. The last layer's activation is a softmax function for classification purposes.

An ANN's input and output are the same as the multiclass logistic regression model, so the categorical cross-entropy loss in equation (5.22) and the gradient term in equation (5.16) are also applicable to the ANN model.

5.3 Implementation

As the software tools for machine learning are becoming increasingly available and versatile, choosing an off-the-shelf software package is more practical and less error-prone than building one from scratch. This section overviews the current popular machine learning tools and presents the diagnosis model structure.

5.3.1 Software Tools

The data processing, model training and testing are implemented in Python. Although there are other programming languages popular in academia, such as C++, R, and Julia, Python remains the most supported choice in terms of third-party libraries. As of writing this dissertation, the mainstream Python library packages for machine learning and deep learning are Scikit-learn, TensorFlow, and PyTorch. All of them are open-sourced.

- Scikit-learn [86] is a high-level machine learning package containing many statistical models, such as regression, classification, clustering, and dimension reduction. It also provides templates and guidelines for customizing models. Many high-quality and useful tools for dataset processing, feature extraction, and test validation are also available in this library. Scikit-learn has been emphasizing machine learning models, although it lacks support for various neural network variations, especially the recent deep learning development. Scikit-learn only supports Python programming language as of writing.
- TensorFlow [87] is a machine learning library with a focus on ANNs. It was developed by Google and later became opensource. There are two major versions. TensorFlow 1.x has established an efficient computation framework for training various neural networks, but its API remains low-level and requires significant machine learning expertise to use. Keras, another library that provides high-level API to developers, was created to act as a user-friendly interface for TensorFlow. TensorFlow 2.x, released in 2019 and the main version available as of writing, integrated Keras within its package and improved its API. TensorFlow excels in

its multi-platform support including mobile and embedded devices, attracting attention from both academia and industry. It is also available in many other programming languages such as JavaScript, R, and Julia.

- PyTorch [88] is a machine learning library developed by Facebook similar to TensorFlow. It is primarily interfaced with Python. PyTorch is simpler, easier to use, and more prevalent in academia in comparison to TensorFlow, but from a high-level perspective, the differences between the two are nuanced.

Scikit-learn is the primary choice for traditional machine learning while TensorFlow or PyTorch are the best candidates for deep learning. There are other libraries such as the Cognitive Toolkit (CNTK) developed by Microsoft and Apache MXNet developed by Amazon, but these are less influential and more related to the companies' own products and services.

Matlab also has its own packages for machine learning and deep learning. It has the advantage of well-maintained help documents and expert support. However, Matlab being a proprietary software lacks the versatility of building a deeply customized model compared to PyTorch and TensorFlow.

For the fault diagnosis model implemented in this chapter, the Scikit-learn package is used for the vectorization process and the TensorFlow 2.x package is used for building and training a model. The program is implemented in Python 3.6 on Windows 10. The computer hardware includes an Intel i7 processor and 32GB memory.

5.3.2 Available Data

The Bluetooth fault dataset comprising log samples and fault labels is the same as the previous chapter. The available fault classes and data samples are shown in Table 5.1. This train-test split setting is also the same as in the previous chapter.

Table 5.1 The available data and train-test split.

Class Label	Fault ID	Total Samples	Samples for Training	Samples for Testing
0	No Identifiable Fault	72	38	34
1	FORDSYNC3-40557	8	4	4
2	FORDSYNC3-40026	3	2	1
3	FORDSYNC3-38112	4	2	2
4	FORDSYNC3-37158	3	2	1
5	FORDSYNC3-32240	4	2	2
6	FORDSYNC3-28906	4	2	2
7	FORDSYNC3-28578	8	4	4
8	FORDSYNC3-28414	8	4	4
	Total	114	60	54

5.3.3 The Diagnosis System Framework

An overview of the FDSML framework is shown in Figure 5.3.1. Each log sample is first converted into a token sequence, a process identical the one in the data mining approach. Then the token sequence goes through the vectorization process to obtain its TF-IDF feature vector as explained in Section 5.2.2.4. There are in total 1482 unique tokens in the dataset, so the length of the TF-IDF vector is 1482. The vectorization converts each token sequence into a 1482×1 vector, denoted by $\boldsymbol{x}^{(i)}$. This feature vector is the input and training data for a machine learning model. The model's output $\boldsymbol{y}^{(i)}$ is a 9×1 vector, each value of $\boldsymbol{y}^{(i)}$ representing the probability of one class label (the regular sample has a label of 0). The highest value of $\boldsymbol{y}^{(i)}$ is the diagnosis result of the model. An MLP as explained in Section 5.2.3.4 is chosen because of its versatility and the complexity of this problem. After the model is trained until its loss value converges, it is evaluated against the test data specified in Table 5.1.

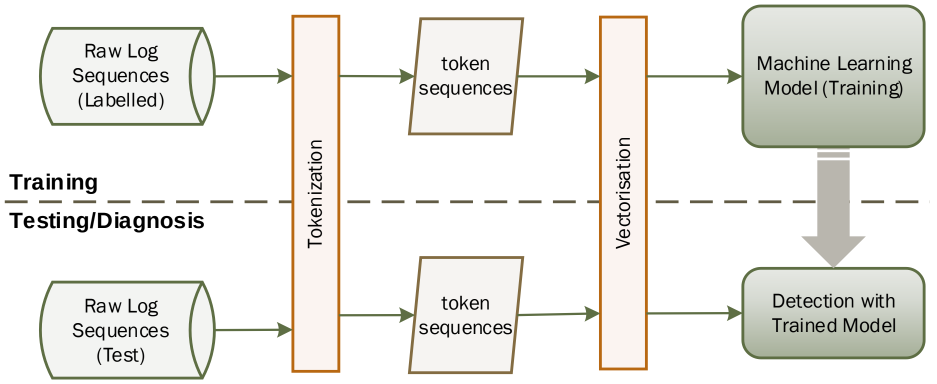


Figure 5.3.1 The structure of fault diagnosis with feature vectorization and machine learning.

5.4 Experiments

In this set of experiments, the detection accuracy and classification accuracy are used for evaluation. Detection means the model correctly identifies a sample as being regular or faulty, i.e., the label 0. Classification means the model correctly produces the fault classes, i.e., the non-zero labels.

The tunable parameters in an MLP include the number of layers, the number of neurons in each layer, and the choice of activation. They are individually examined in the following subsections.

5.4.1 The Training Process

Training is the process of a machine learning model adjusting its weights to fit the labelled data. Specifically, the training dataset is fed into the model repeatedly, while an optimisation algorithm, such as the SGD, updates the weights iteratively. Processing one cycle of the training data is called one *epoch*. The loss value is expected to decrease over epochs. The model's accuracy on training samples should have an opposite trend to the loss curve. The training should be stopped at the time when the loss converges.

When training a machine learning model, especially ANNs, the issue of underfitting or overfitting can occur. Overfitting is the phenomenon when a model fits extremely well training data, but loses generalization over new or unseen data. An overfitting model produces lower accuracies on the test data than the training data. Conversely, underfitting is when a model cannot generalize well on the training data and the loss converges at a high value. An underfitting model often produces higher accuracies on the test data than the training data. Both issues are caused by the inadequate structure and sizing of the network. In general, a slightly overfit model is preferred over an underfitting one. This is because overfitting can be suppressed by specifying early stopping of the training, while underfitting can only be addressed by reconfiguring the model structure. Ideally, an additional validation dataset can be used to make the underfitting or overfitting easier to observe. However, due to the lack of available data, this study does not adopt a validation set. The test data in this experiment section is used effectively as the validation data.

As a demonstration, the training process of a 3-layer MLP is shown below. Table 5.2 shows the configuration of the network comprising an input layer, one hidden layer, and an output layer. The sizes of the input and the output layers are specified by the size of the sample vector and the number of class labels, respectively. The size of the hidden layer is chosen arbitrarily as 64.

Table 5.2 The configuration of a 3-layer MLP.

Layer	Configuration	Output Shape	Number of Parameters
Input	None	1482	0
hidden_1	size=64, sigmoid	64	94,912
Output	size=9, softmax	9	585
		Total Parameters:	95,497

The network was trained using SGD. The loss value and accuracy during the training process are shown in Figure 5.4.1. As expected, the loss value has a decreasing trend until settling to a small value. The accuracy has an opposite trend and settles at 98%. It takes 1500 epochs and 87 seconds to reach convergence. Note that the training can keep on to drive the loss even lower, but it is unnecessary as the accuracy has plateaued.

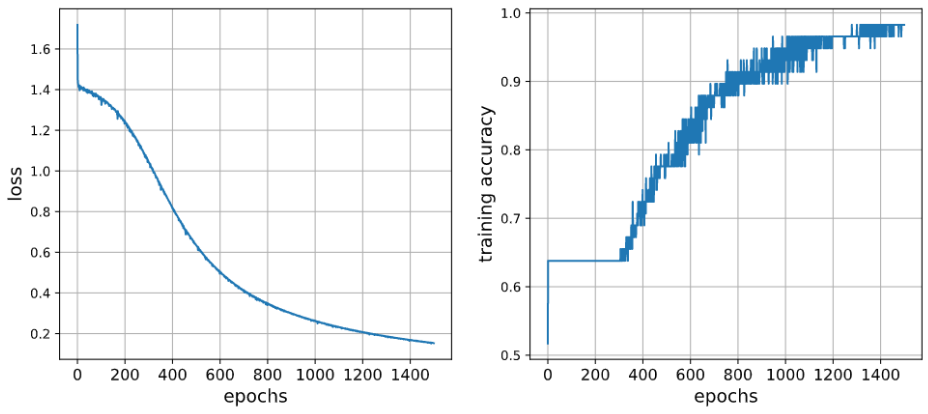


Figure 5.4.1 Loss and accuracy during the training process.

The test accuracy was produced by evaluating the model on the test dataset. The test accuracy for this trained model was 60.7%, significantly lower than the training accuracy. This indicates an overfitting issue. Therefore, a training accuracy threshold was set to stop the training in an early stage and alleviate overfitting.

5.4.2 Early Stopping of the Training

In a separate training experiment, a threshold of 70% was selected to be slightly higher than the test accuracy. The training process is shown in Figure 5.4.2. This time it only takes 350 epochs to reach target accuracy. The new model produced a test accuracy of 62.5%, which is better than the highly overfit model. This demonstrates that overfitting can lead to loss of generality and lower performance than the model should be.

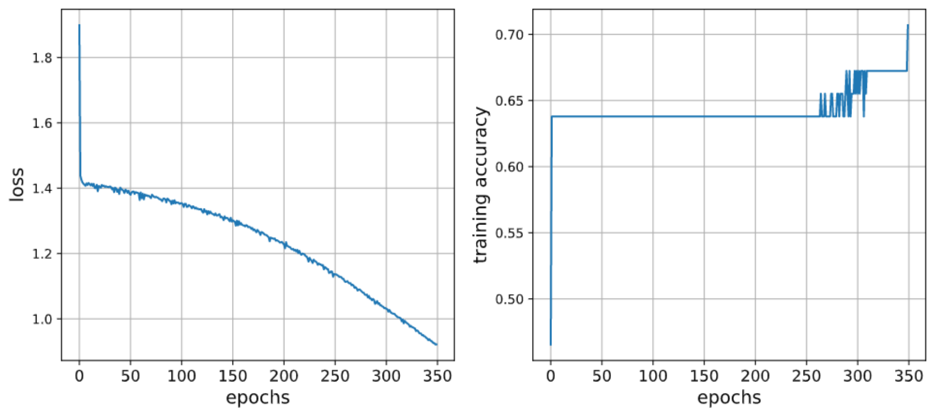


Figure 5.4.2 Training loss and accuracy with an early-stopping threshold of 70%.

5.4.3 Batch Gradient Descent

In addition to the SGD that uses the gradient with regard to one sample to update the network weights, batch gradient descent using the average gradient of a batch of samples is also tested. Table 5.3 shows the time and number of epochs needed to train the model to a training accuracy of 70%. The number of epochs grows with the size of the batches, but the training time shows a slight increase. This means for larger batch sizes, each training epoch costs less amount of time. The time reduction of each epoch is because the batch algorithm processes multiple samples at a time. However, the acceleration of each epoch has little contribution to the overall training time. Therefore, SGD is selected as the default training method.

Table 5.3 The effect of batch gradient descent.

Batch size	**Number of epochs to reach 70% training accuracy**	**Training Time to reach 70% training accuracy**
1 (SGD)	~350	~20s
2	~750	~22s
4	~1500	~26s
8	~2600	~26s

Another character brought by the batch processing is smoothing out the loss curve. The training curves with a batch size of 8 are shown in Figure 5.4.3. Comparing with Figure 5.4.2, the most noticeable difference is the smoothed-out loss curve. Batch gradient descent could be beneficial when the training curve contains too much clutter that hinders convergence, but this is not an issue for this study. Therefore, the rest of experimentation would use the SGD without batch processing.

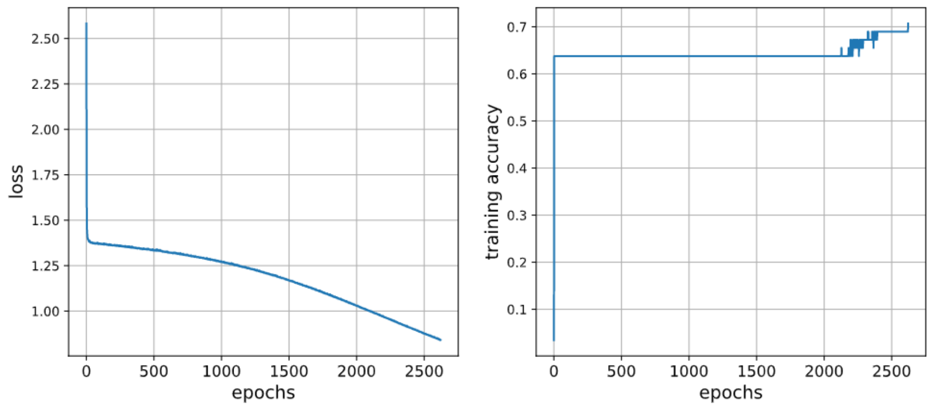


Figure 5.4.3 Training loss and accuracy with a training batch size of 8.

5.4.4 The Size of the Network

The number of layers and the number of neurons within each layer are configurable to accommodate the complexity of the problem. In this set of experiments, three layer settings and multiple neuron settings are tested to examine how network size affects performance. The models' configurations are shown in Table 5.4. The layer settings start from 2. A 2-layer neural network contains only an input layer and an output layer, with no hidden layer in-between, making it effectively a multiclass logistic regression model. The 3-layer and 4-layer models contain one and two hidden layers, respectively. The number of neurons in the hidden layer refers to the size of the hidden layer. This method of finding the best configuration is called grid search. In total, there are 14 configurations.

Table 5.4 MLP models with different sizes.

	2-layer (Multiclass logistic regression)	**3-layer**	**4-layer**
Hidden layer size	N/A	32 64 128 256	16, 16 16, 32 16, 64 32, 16 32, 32 32, 64 64, 16 64, 32 64, 64

The metric to evaluate the performance is the detection accuracy and classification accuracy, same as the previous chapter. Detection accuracy is defined by the number of correct detections divided by the total number of samples. Classification accuracy is the number of correctly classified faults divided by the total number of faulty samples. The performance of all 14 models evaluated on the test set is shown in Table 5.5. Each entry in this table takes an average of 12 individual runs to account for the randomness during the train-test split and network initialization processes.

Table 5.5 The performance of models with different configurations.

Model ID	Total Number of Layers	Size of Hidden Layer	Detection Accuracy (average)	Classification Accuracy (average)
1	2	N/A	60.7%	46.7%
2	3	32	59.0%	47.1%
3	3	64	60.3%	48.8%
4	3	128	60.7%	46.7%
5	3	256	62.8%	45.8%
6	4	16, 16	58.3%	50.0%
7	4	16, 32	57.7%	50.0%
8	4	16, 64	55.8%	47.5%
9	4	32, 16	57.1%	47.5%
10	4	32, 32	58.3%	48.7%
11	4	32, 64	56.4%	46.2%
12	4	64, 16	58.3%	47.5%
13	4	64, 32	57.7%	47.5%
14	4	64, 64	57.7%	46.2%

According to Table 5.5, a 3-layer MLP with 256 neurons in the hidden layer (Model No.5) achieves the highest detection accuracy, while a 4-layer MLP with both hidden layers' sizes of 16 gets the best classification accuracy. It can be hard to directly observe the trend of the other rows in such a large table, so a scatter plot is produced in Figure 5.4.4. The x and y coordinates of each dot represent a model's performance in terms of detection and classification accuracy, respectively. The dots closer to the upper right corner are preferred. From the clusters, using the 4-layer setting tends to produce lower detection accuracy, although some configurations have better classification. The 3-layer setting has an opposite tendency. A balance between both metrics is the 3-layer setting with a hidden size of 64 (Model No.3) and the 4-layer setting with both hidden sizes of 16

(Model No.6). Therefore, these two configurations should be selected as the machine learning model for this fault diagnosis application.

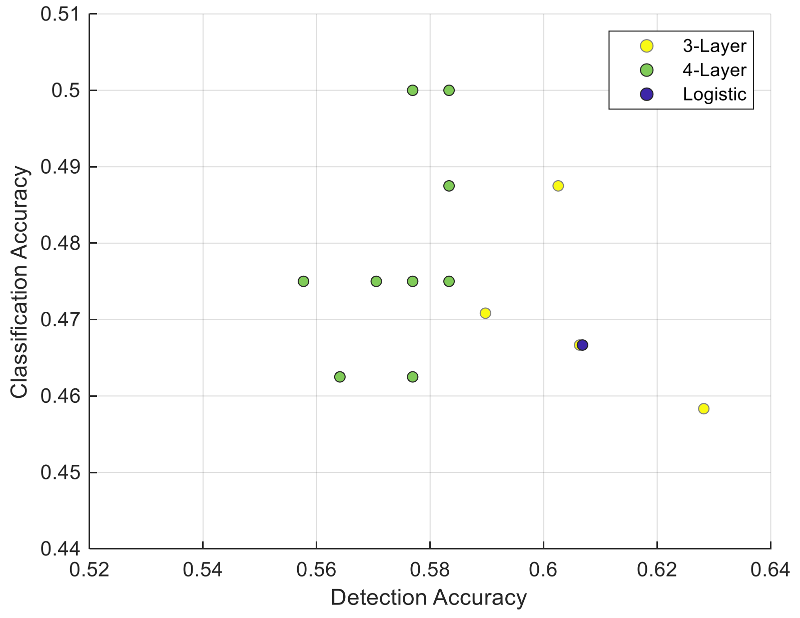


Figure 5.4.4 The performance of MLP models with different configurations.

5.4.5 Discussion

Comparing with the data-mining approach introduced in the previous chapter, the biggest advantage of using statistical features and an MLP model is the efficiency. Each training process in this section takes less than one minute, whereas the sequence feature discovery in the previous chapter took on average two hours. The saved time means that shorter and easier parameter tuning for the training process. When deployed, this machine learning method will impose a shorter computation overhead such that online diagnosis could be possible.

However, the performance metrics of the MLP model are relatively lower than the previous naïve Bayes classifier, which produces around 75% detection and 60%

classification accuracy. This may be due to the lack of training data. An MLP contains thousands of trainable parameters to converge. The fact that each fault class only contains an average of three training samples is far from adequate. The lack of training data likely caused the model to converge on an ill-defined optimal point. The data-mining method on the other hand used contrast mining approach as a way of enriching the training data, reduced the requirement for training samples.

To evaluate a properly trained MLP model's performance, artificially augmented log data with sufficient samples are created. The next section presents how this dataset is created and how the model performs on the dataset.

5.5 Experiments Using Augmented Data

The artificially created dataset is to evaluate a model's performance in the real-world scenario, therefore the dummy samples in the dataset must be as close to the original log as possible. The samples are made by injecting fault patterns into normal log sequences, both of which are from real log data. Two fault classes are selected with fault IDs FORDSYNC3-28507 (fault 1) and FORDSYNC3-17456 (fault 2), representing the Bluetooth connection pairing failure and the Bluetooth fatal error, respectively. Their faulty patterns are confirmed by subject matter experts. The patterns are inserted randomly into normal sequences to create a faulty sequence sample, as shown in Figure 5.5.1. The original data segments in the figure are log sequences from normal system execution containing no known faults.

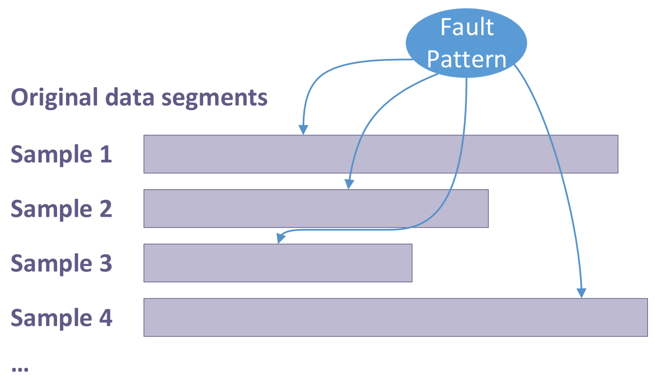


Figure 5.5.1 Creating artificially injected fault samples.

The data to construct the artificial dataset are the 72 samples with label 0 in Table 5.1. Their sequences are randomly broken down into shorter segments. Each segment has an average length of around 500, which is significantly longer than the fault patterns. This creates 370 dummy samples, every one of them is unique and authentic. Then one-third are injected with the patterns for fault 1, one-third are injected with fault 2, and the rest are used as normal samples with class label 0. The artificial dataset is split into a training set and a test set with a 50:50 ratio.

5.5.1 Evaluation Using the Artificial Dataset

The previous section has concluded that a 3-layer MLP is a good candidate for the scale of this diagnosis problem, so a 3-layer MLP is used as the model for this evaluation using the artificial dataset. Three sizes of the hidden layer – 64, 128, and 256 – are tested in order to find a better model configuration. Similar to the previous experiments, each training is individually performed 12 times to take account of the random variations in the training process. All the models are trained until the loss converges.

The average performance of these models on the test set is shown in Table 5.6. All three models achieved over 80% accuracies on both detection and classification tasks, a boost of almost 30% greater than the models trained in the previous section. Models with larger hidden layer sizes can push the detection accuracy closer to 90%, while the classification accuracy remains around 85%. These results indicate that a properly trained model can be very effective at diagnosing faulty samples that contain faulty sequence patterns. A larger network tends to provide better detection, while the classification is less affected.

Table 5.6 The MLP performance trained on the artificial dataset.

Model ID	Size of Hidden Layer	Detection Accuracy (average)	Classification Accuracy (average)
1	64	83.78%	86.18%
2	128	87.39%	85.37%
3	256	88.65%	85.37%

5.5.2 Results with Reoccurring Patterns

Sometimes the fault pattern can occur multiple times in a faulty sample, as observed from the historical log data. To simulate this scenario, a new dummy dataset is created with each fault pattern injected twice for every sample. The new dataset is used to train and evaluate the same MLP models and the test settings are the same as the previous subsection.

These models show a higher accuracy around and above 90% both in terms of detection and classification, as shown in Table 5.7. This is a significant increase from the previous test. The performance shows a positive correlation with the size of the network. The largest network with a hidden layer of 256 achieved the best results with a detection accuracy of 91.89% and a classification accuracy of 90.24%. These results demonstrate that the MLP models are more effective when the fault patterns have multiple occurrences, confirming the capability of these models.

Table 5.7 The MLP performance trained on the artificial dataset with multiple fault pattern occurrences.

Model ID	Size of Hidden Layer	Detection Accuracy (average)	Classification Accuracy (average)
4	64	90.09%	89.97%
5	128	90.81%	89.70%
6	256	91.89%	90.24%

5.6 Conclusion

This section presents an automated software diagnosis framework, the FDSML, that incorporates the statistical feature vectorization of the log sequences and machine learning models. The vectorization module uses numerical features to describe log samples. The machine learning model processes the feature vectors and produces possible fault classes with probability ratings. The framework is versatile in terms of the choice of vectorization and machine learning methods, as long as they are numerical methods.

In particular, the TF-IDF feature vectorization, the logistic regression model, and the MLP model are developed and implemented. The detection system is trained and evaluated using the Ford SYNC log dataset. A grid search is implemented to find the optimal network configuration for the MLP model.

The MLP model trained on the SYNC dataset shows some effectiveness in terms of detection and classification accuracy, however it is lower than the data mining approach. The issue is found to be due to the lack of training data. Additional tests show that the MLP models exhibit a 30% accuracy improvement when trained on an artificially enriched dataset. Multiple occurrences of the fault patterns would drive the models' performance even higher.

Computational efficiency is the biggest advantage of FDSML framework comparing to the data mining approach in the previous chapter. With the training time only a fraction of the previous method, the application of the new system is capable of including a large amount of data, reducing the effort of tuning, and possibly applying to online detection and diagnosis.

One drawback of this approach is within the vectorization process, which focuses on the statistical representation of individual tokens. Such vectorization mostly disregards the sequential order and timestamp information. The next chapter will explore the deep learning methodology that addresses these concerns.

Chapter 6 Deep Learning Methods

This chapter presents a deep learning approach to the automated log analysis and fault diagnosis problem. A deep learning model is similar to a traditional machine learning one in the way that both use numerical representations of the log samples to perform classification tasks. Instead of extracting sequence-based features like the TF-IDF vector introduced in the previous chapter, the deep learning approach learns the token-based features. This preserves the sequential information and results in large feature matrices to represent token sequences. To fit these features requires a specialized neural network model with potentially a large number of layers – a deep learning model. This deep learning approach is able to extract and process more information from the log data than a traditional machine learning one.

Deep learning, or deep neural networks, is a subset of machine learning based on artificial neural networks. A Multi-Layer Perceptron (MLP) model is often referred to as a shallow neural network, as it typically comprises a small number of layers. The total number of parameters in an MLP quickly adds up as the size of the network grows. This causes the training to be inefficient and easily overfit. Another drawback of an MLP is that it only processes input in a vector format. Recent deep learning development includes new network architectures such as convolutional neural networks and recurrent neural networks, which allow deeper networks – more layers without compromising training – and adapt to various types of input data. Many areas have seen successful applications of using the deep learning models to perform classification tasks, such as image recognition and natural language text comprehension.

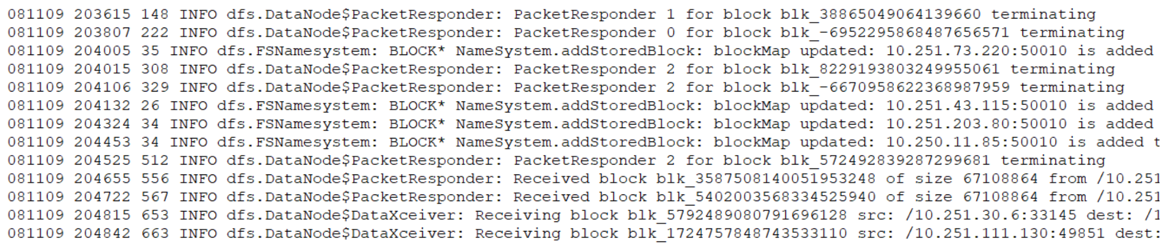
Previous chapters have concluded that the lack of data being the bottleneck of machine learning based detection and diagnosis systems. In this chapter, open datasets are considered for training and evaluation of the deep learning system [89]. The Hadoop distributed file system (HDFS) is one that produces similar unstructured logs as the Ford system. HDFS is a data storage solution that manages data on cluster machines. The dataset collects the runtime log of the HDFS and is manually labelled as normal or faulty. The amount of data is abundant, so it is ideal for training and evaluating a deep learning model. In fact, a few studies on log analysis have been using this dataset, including machine learning methods and deep learning. The HDFS dataset is introduced in Section 6.1.

In this study, two deep learning models are proposed for fault detection using system log data. An overview of the system framework is introduced in Section 6.2. The embedding process that numerically represents the tokens and sequences is explained in Section 6.3. Section 6.4 elaborates the details of two deep learning classification models. The model implementation and evaluation are presented in Sections 6.5 and 6.6, respectively. Section 6.7 concludes this chapter.

6.1 The HDFS Dataset and Data Preparation

The Hadoop distributed file system (HDFS) is a data storage and management system that runs on a cluster of computers. The cluster comprises a server – formally known as the NameNode – and a large number of data-storing computing blocks – formally known as the DataNode. When a request is sent to the server, it communicates with its fellow data blocks to perform certain tasks. This process creates log information. Failures can happen within the computing blocks, causing the system execution to enter a faulty state. The log messages related to a certain block effectively create a log sequence, which could be used to infer the health condition of the block. This resembles the Ford log as the SYNC system has multiple modules, a fault would occur to one module, resulting in abnormal behaviour of the log sequences related to that module.

The HDFS log benchmark dataset was originally introduced in [29] and openly available from [90]. The healthy status of each data block is labelled as normal or faulty. The original form of this dataset is a single 1.47 GB text file with 11.2 million log lines, recording 38.7 hours of the HDFS system runtime. The log lines are essentially unstructured text. Figure 6.1.1 shows a screenshot of a few log lines from the original data file. The fields in each long line include the timestamp, message ID, message category, and a descriptive statement. Unlike the Ford logs, the HDFS logs do not contain an explicit identifier (such as the token value introduced in Section 4.2.2) that can be used to distinguish and tokenize log messages. Therefore, a rather complicated parsing process is required before the dataset can be used for training.



```
081109 203615 148 INFO dfs.DataNode$PacketResponder: PacketResponder 1 for block blk_38865049064139660 terminating
081109 203807 222 INFO dfs.DataNode$PacketResponder: PacketResponder 0 for block blk_-6952295868487656571 terminating
081109 204005 35 INFO dfs.FSNamesystem: BLOCK* NameSystem.addStoredBlock: blockMap updated: 10.251.73.220:50010 is added
081109 204015 308 INFO dfs.DataNode$PacketResponder: PacketResponder 2 for block blk_8229193803249955061 terminating
081109 204106 329 INFO dfs.DataNode$PacketResponder: PacketResponder 2 for block blk_-6670958622368987959 terminating
081109 204132 26 INFO dfs.FSNamesystem: BLOCK* NameSystem.addStoredBlock: blockMap updated: 10.251.43.115:50010 is added
081109 204324 34 INFO dfs.FSNamesystem: BLOCK* NameSystem.addStoredBlock: blockMap updated: 10.251.203.80:50010 is added
081109 204453 34 INFO dfs.FSNamesystem: BLOCK* NameSystem.addStoredBlock: blockMap updated: 10.250.11.85:50010 is added t
081109 204525 512 INFO dfs.DataNode$PacketResponder: PacketResponder 2 for block blk_572492839287299681 terminating
081109 204655 556 INFO dfs.DataNode$PacketResponder: Received block blk_3587508140051953248 of size 67108864 from /10.251
081109 204722 567 INFO dfs.DataNode$PacketResponder: Received block blk_5402003568334525940 of size 67108864 from /10.251
081109 204815 653 INFO dfs.DataNode$DataXceiver: Receiving block blk_5792489080791696128 src: /10.251.30.6:33145 dest: /1
081109 204842 663 INFO dfs.DataNode$DataXceiver: Receiving block blk_1724757848743533110 src: /10.251.111.130:49851 dest:
```

Figure 6.1.1 Raw log data in text format.

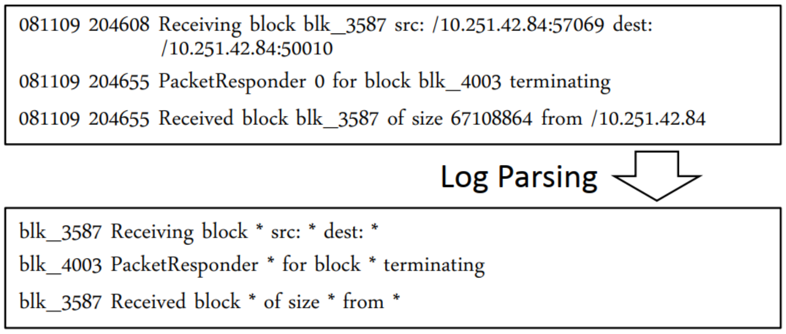
6.1.1 Parsing

The log messages contain valuable related information, such as the block ID and event types in the descriptive statement by the end of each line, but they are difficult for a machine to interpret. These statements are meant for human developers and come in various formats or templates that are not evident. Interpreting this part of log messages is necessary for creating sequence samples and tokenization.

Extracting useful information from the descriptive statement and sorting them into categories is a topic called log parsing. A few researchers have well studied this topic and achieved good parsing accuracy [37], [56], [61], [91]–[94]. In particular, the Drain algorithm [92] uses a tree structure to parse the event types in an unsupervised manner. This method is chosen for preprocessing because of its accuracy and efficiency demonstrated in an evaluation study [95].

From a high-level overview, the Drain method creates a fixed depth tree using all available log statements. The tree's leaf nodes represent a log statement template. The tree's internal nodes contain specially designed rules to branch out the search, such as the number of words in the statement, the starting word, and the second word. If a log statement matches the branches and successfully traverses to a leaf node, it belongs to the template represented by the leaf. Otherwise, a new branch is added to the tree, creating a new template.

A general description of the parsing process is shown in Figure 6.1.2. The top box lists three log messages, and the bottom box shows the correct parsing result. The main output is the template that represents log statement. For example, the first row in the bottom box extracts the template that matches the first log message and leaves variables as placeholders indicated by start sign *. This template, or specifically an event template, is the main output of the parsing process. Another output is the string "blk_****". It represents the physical element (a data block) that can be either regular or faulty.


```
081109 204608 Receiving block blk_3587 src: /10.251.42.84:57069 dest:
              /10.251.42.84:50010
081109 204655 PacketResponder 0 for block blk_4003 terminating
081109 204655 Received block blk_3587 of size 67108864 from /10.251.42.84
```

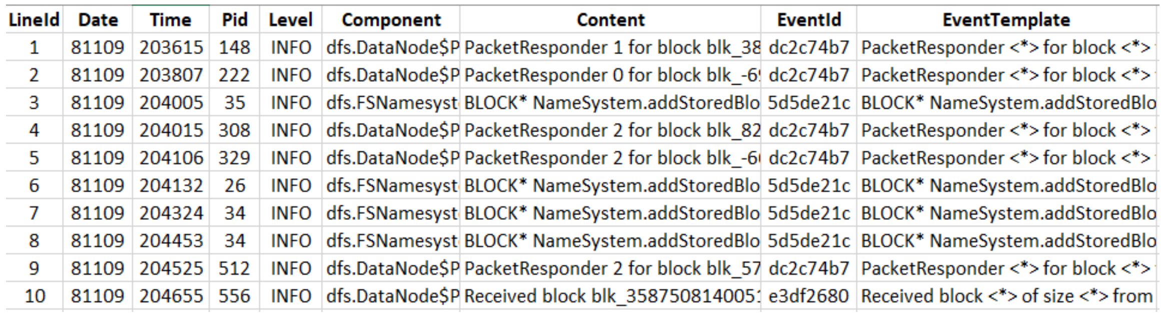
```
blk_3587 Receiving block * src: * dest: *
blk_4003 PacketResponder * for block * terminating
blk_3587 Received block * of size * from *
```

Figure 6.1.2 Overview of Parsing HDFS logs using the Drain method. [92]

A detailed parsing result is shown in Figure 6.1.3. It shows ten parsed log messages in a structured format. The columns listed in the figure are as follows.

- Date, Time, Pid, Level, Component are directly obtained from the beginning of a log message shown in Figure 6.1.1.
- Content is the descriptive statement, the second half of a log message.

- EventTemplate is template representing a logging event. It is extracted by the Drain method explained in the previous paragraph.
- EventId is the unique ID for each EventTemplate. It can be treated as the token of the log message.
- Another critical piece of information, the block ID, a string “blk_****” extracted from the Content column using a regular expression match.



LineId	Date	Time	Pid	Level	Component	Content	EventId	EventTemplate
1	81109	203615	148	INFO	dfs.DataNode$P	PacketResponder 1 for block blk_38	dc2c74b7	PacketResponder <*> for block <*>
2	81109	203807	222	INFO	dfs.DataNode$P	PacketResponder 0 for block blk_-6	dc2c74b7	PacketResponder <*> for block <*>
3	81109	204005	35	INFO	dfs.FSNamesyst	BLOCK* NameSystem.addStoredBlo	5d5de21c	BLOCK* NameSystem.addStoredBlo
4	81109	204015	308	INFO	dfs.DataNode$P	PacketResponder 2 for block blk_82	dc2c74b7	PacketResponder <*> for block <*>
5	81109	204106	329	INFO	dfs.DataNode$P	PacketResponder 2 for block blk_-6	dc2c74b7	PacketResponder <*> for block <*>
6	81109	204132	26	INFO	dfs.FSNamesyst	BLOCK* NameSystem.addStoredBlo	5d5de21c	BLOCK* NameSystem.addStoredBlo
7	81109	204324	34	INFO	dfs.FSNamesyst	BLOCK* NameSystem.addStoredBlo	5d5de21c	BLOCK* NameSystem.addStoredBlo
8	81109	204453	34	INFO	dfs.FSNamesyst	BLOCK* NameSystem.addStoredBlo	5d5de21c	BLOCK* NameSystem.addStoredBlo
9	81109	204525	512	INFO	dfs.DataNode$P	PacketResponder 2 for block blk_57	dc2c74b7	PacketResponder <*> for block <*>
10	81109	204655	556	INFO	dfs.DataNode$P	Received block blk_358750814005	e3df2680	Received block <*> of size <*> from

Figure 6.1.3 Structured log data after parsing.

As a result of parsing, a log message is represented by three fields:

- the timestamp, a combination of the “Date” and “Time” columns.
- block ID, extracted from the “Content” column.
- event type (token), the “EventId” column.

Figure 6.1.4 shows the final parsing result using the three fields for one log message. In this example, the token “d38aa58d” represents the event described in the original log statement, and this event occurred at the data block indicated by the block ID. Apart from these three fields, the rest of the columns in Figure 6.1.3 are removed as they are either duplicates or providing little useful information.

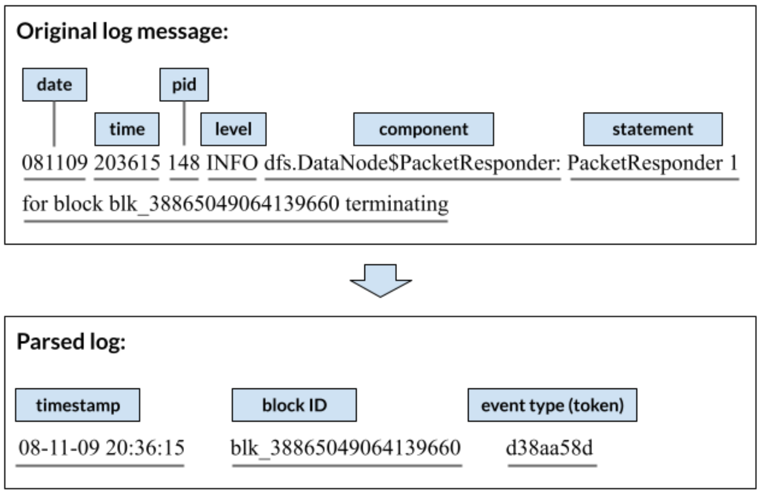


Figure 6.1.4 An example of parsing the HDFS log message.

6.1.2 Sequencing

Since the failures in the HDFS system are associated with data blocks, logs events that have the same block ID are grouped. An example of the grouping is shown in Figure 6.1.5. The log events in this group effectively form sequences: one token sequence and one timestamp sequence, and the block ID represents the sequence being formed. As expected, the total number of sequences equals the number of blocks in an HDFS system.

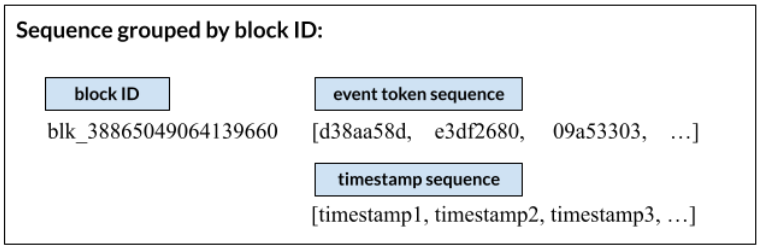
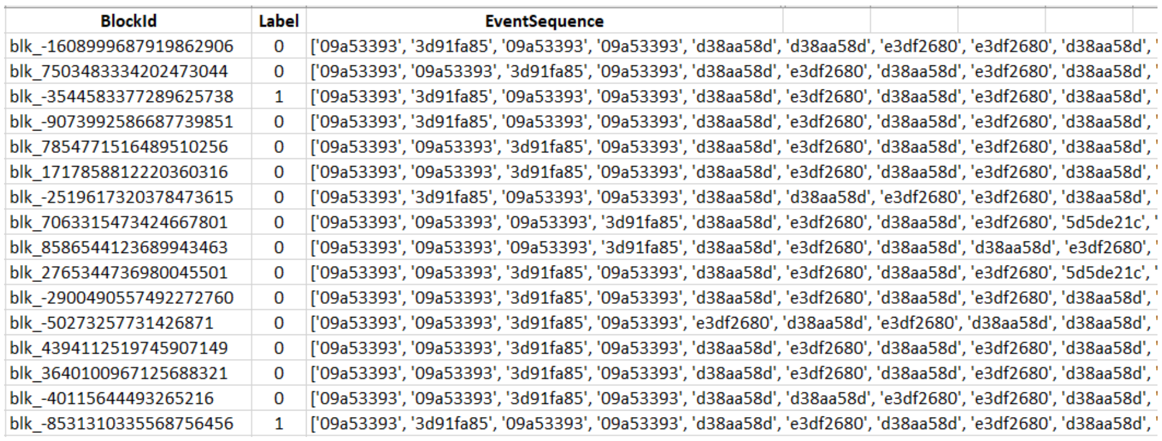

Sequence grouped by block ID:
block ID
blk_38865049064139660
event token sequence
[d38aa58d, e3df2680, 09a53303, …]
timestamp sequence
[timestamp1, timestamp2, timestamp3, …]

Figure 6.1.5 An example of creating log sequences.

Each sequence is then labelled as either 0 or 1, corresponding to the state of the block of regular or faulty. This labelling information is available from the HDFS log dataset. A snapshot of the sequencing outcome is shown in Figure 6.1.6. The columns list the block ID, its label, and the event token sequence.



BlockId	Label	EventSequence
blk_-1608999687919862906	0	['09a53393', '3d91fa85', '09a53393', '09a53393', 'd38aa58d', 'd38aa58d', 'e3df2680', 'e3df2680', 'd38aa58d', '
blk_7503483334202473044	0	['09a53393', '09a53393', '3d91fa85', '09a53393', 'd38aa58d', 'e3df2680', 'd38aa58d', 'e3df2680', 'd38aa58d', '
blk_-3544583377289625738	1	['09a53393', '3d91fa85', '09a53393', '09a53393', 'd38aa58d', 'e3df2680', 'd38aa58d', 'e3df2680', 'd38aa58d', '
blk_-9073992586687739851	0	['09a53393', '3d91fa85', '09a53393', '09a53393', 'd38aa58d', 'e3df2680', 'd38aa58d', 'e3df2680', 'd38aa58d', '
blk_7854771516489510256	0	['09a53393', '09a53393', '3d91fa85', '09a53393', 'd38aa58d', 'e3df2680', 'd38aa58d', 'e3df2680', 'd38aa58d', '
blk_1717858812220360316	0	['09a53393', '09a53393', '3d91fa85', '09a53393', 'd38aa58d', 'e3df2680', 'd38aa58d', 'e3df2680', 'd38aa58d', '
blk_-2519617320378473615	0	['09a53393', '3d91fa85', '09a53393', '09a53393', 'd38aa58d', 'd38aa58d', 'e3df2680', 'e3df2680', 'd38aa58d', '
blk_7063315473424667801	0	['09a53393', '09a53393', '09a53393', '3d91fa85', 'd38aa58d', 'e3df2680', 'd38aa58d', 'e3df2680', '5d5de21c', '
blk_8586544123689943463	0	['09a53393', '09a53393', '09a53393', '3d91fa85', 'd38aa58d', 'e3df2680', 'd38aa58d', 'd38aa58d', 'e3df2680', '
blk_2765344736980045501	0	['09a53393', '09a53393', '3d91fa85', '09a53393', 'd38aa58d', 'e3df2680', 'd38aa58d', 'e3df2680', '5d5de21c', '
blk_-2900490557492272760	0	['09a53393', '09a53393', '3d91fa85', '09a53393', 'd38aa58d', 'e3df2680', 'd38aa58d', 'e3df2680', 'd38aa58d', '
blk_-50273257731426871	0	['09a53393', '09a53393', '3d91fa85', '09a53393', 'e3df2680', 'd38aa58d', 'e3df2680', 'd38aa58d', 'd38aa58d', '
blk_4394112519745907149	0	['09a53393', '09a53393', '3d91fa85', '09a53393', 'd38aa58d', 'e3df2680', 'd38aa58d', 'e3df2680', 'd38aa58d', '
blk_3640100967125688321	0	['09a53393', '09a53393', '3d91fa85', '09a53393', 'd38aa58d', 'e3df2680', 'd38aa58d', 'e3df2680', 'd38aa58d', '
blk_-40115644493265216	0	['09a53393', '09a53393', '3d91fa85', '09a53393', 'd38aa58d', 'd38aa58d', 'e3df2680', 'e3df2680', 'd38aa58d', '
blk_-8531310335568756456	1	['09a53393', '3d91fa85', '09a53393', '09a53393', 'd38aa58d', 'e3df2680', 'd38aa58d', 'e3df2680', 'd38aa58d', '

Figure 6.1.6 Sequenced and labelled log data.

Furthermore, the block ID column in Figure 6.1.6 is discarded as it is not relevant to the detection system. The label and token sequence are used as samples for training and evaluation purposes. The timestamp sequence is set aside for now – the next chapter will discuss the integration of timestamps.

6.1.3 Overview of the Sequence Dataset

After parsing and sequencing, the original 11.2-million-line log file is converted into a dataset containing 575,061 labelled sequence samples. These sequences consist of 48 unique tokens, representing 48 different software events. A majority of these samples have lengths between 10 to 40, with an average of 19.43. The histogram of sequence length is shown in Figure 6.1.7. Some outliers with lengths up to 298 are not displayed in this graph.

The dataset is an imbalanced one containing more normal samples than faulty ones. The number of faulty samples is 16,838, less than 3% of the total samples. Such a dataset is called an imbalanced dataset. The histograms of faulty and normal samples are shown in Figure 6.1.8. One interesting observation is that the short samples with a length less than 10 are most likely to be faulty. For sample lengths of more than 10, there is little distributive difference between faulty and normal datasets. Although the short samples could be easily filtered out by a rule-based detector, this study did not exclude them in order to assess the machine-learning-based models in a comprehensive way.

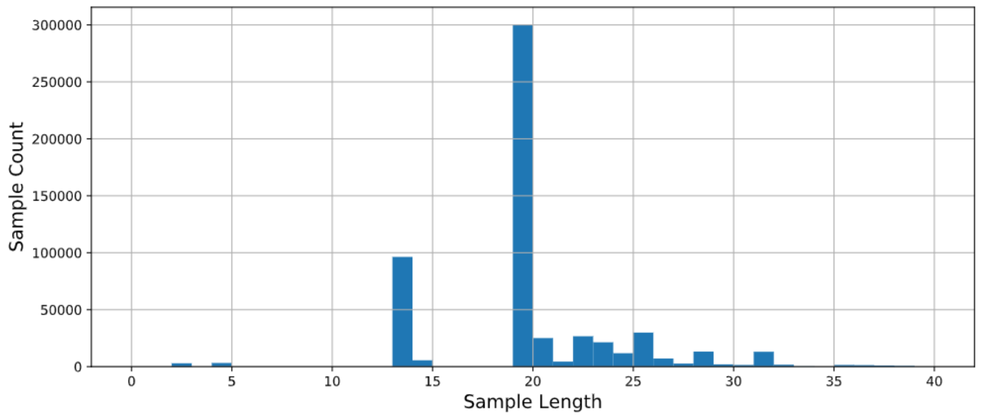


Figure 6.1.7 Histogram of sequence lengths (full dataset).

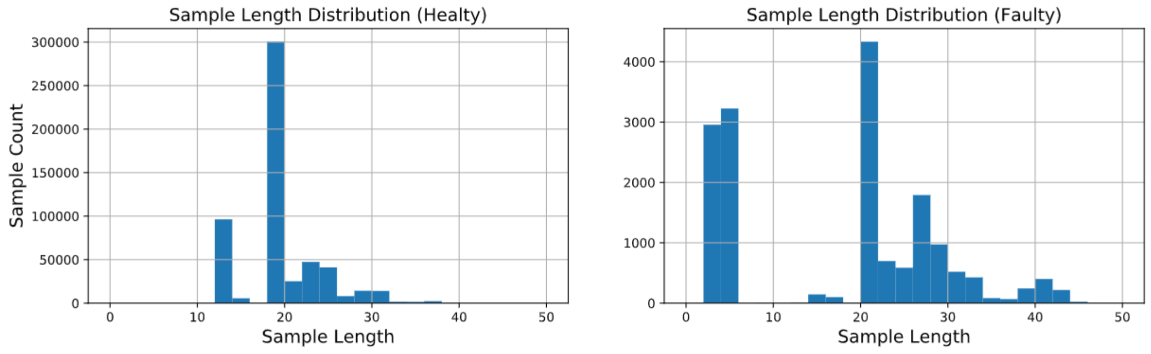


Figure 6.1.8 Histogram of sequence lengths by label.

6.2 The Failure Detection Framework

The automated failure detection and fault diagnosis framework is shown in Figure 6.2.1. As a typical machine learning system, it contains a training phase as indicated on the upper half and a detection/diagnosis phase as shown on the lower half. The raw log data in text format, whether they are from SYNC or HDFS system, first go through the parsing process to obtain the token sequence samples. A deep learning model directly processes the token sequences for training or diagnosis. Note that there is no need for a dedicated preprocessing step, such as the feature vectorization process in the FDSML framework introduced in the previous chapter. Instead, integrated within the deep learning model is an *embedding* process which converts token sequences into numerical representations. The classification part of the deep learning model handles the numerical representations and produces an output. The embedding and classification model are introduced in Sections 6.3 and 6.4, respectively.

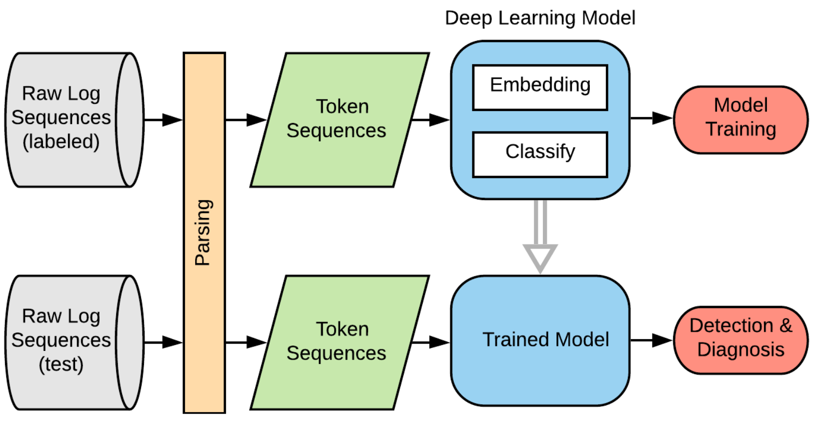


Figure 6.2.1 The overview of deep learning detection and diagnosis system.

6.3 Sequence Embedding

The log samples in the token sequence form need to be represented in numerical form in order for a machine learning or deep learning model to process. In the previous chapter, various vectorization methods were introduced to extract manually defined feature vectors from a sequence. In the deep learning models, the conversion to vectors is still necessary, but there are two differences: 1) the process applies to individual tokens instead of a whole sequence, 2) the features are obtained through self-learning methods, instead of manually defined. This conversion to numeric process is called *embedding*. The embedding represents a sequence in a 2D matrix and preserves the sequential information in the meantime. The following subsections elaborate on this method and its implementation as part of the neural network model.

6.3.1 One-hot Encoding

Encoding for symbolic data refers to the process of converting symbols (tokens) into numerical vectors. The term can be used to refer to the process of conversion as well as the result of the conversion. One-hot encoding is a simple form of encoding. Similar to the one-hot vectorization introduced in Chapter 5, the length of the encoded vector equals the total number of unique tokens. The term one-hot means only one value is 1 and all others are 0 in the vector. Take the example of token sequence “ABCDAE”, the second columns of Table 6.1 shows the one-hot encoding of each token of the sequence. The encoding vector’s shape is 1×5 as there are 5 different tokens in this set. Note that the token “A” appears twice in the sequence and the two appearances both have the same encoding. The converted vectors are then stacked in the same order as the original sequence. Therefore, the encoding of the sequence is effectively a 6×5 matrix where the progression of rows represents the order of the sequence.

Table 6.1 Encoding of the sentence.

Sequence	One-hot Encoding	Reduced Encoding with Autoencoder
A	[1 0 0 0 0]	[2.2 1.9]
B	[0 1 0 0 0]	[-2.2 -0.1]
C	[0 0 1 0 0]	[1.7 -1.1]
D	[0 0 0 1 0]	[-0.9 2.6]
A	[1 0 0 0 0]	[2.2 1.9]
E	[0 0 0 0 1]	[-1.1 -2.9]

One-hot encoding is simple to implement and interpret. However, its disadvantages are also clear: the encoded matrix can get very large and sparse. The HDFS log data in this study has 48 unique tokens; the Ford log contains more than a thousand. The inflated size takes more memory and demands more computational resources, making the network training less efficient. Another limitation is that one-hot encoding assumes no correlation between different tokens, i.e., any two different encoding vectors are orthogonal. The encodings are merely to differentiate the tokens rather than associate them.

For the sparsity problem, various dimensionality reduction techniques are applicable. Autoencoder is one of the learning-based candidates. In its simplest form, an Autoencoder is a neural network that tries to replicate the input as the output. Its input side is purposely made symmetrical to the output, as shown in Figure 6.3.1. The network has an odd number of hidden layers whose sizes are typically smaller than the input or output layers. Once trained, the output values are identical or very close to input values. As a feedforward neural network, knowing the values of any hidden layer can propagate forward to obtain the outputs that approximate the input. In other words, every hidden layer contains all the information to reconstruct the input. The smallest hidden layer, which is often the layer in the middle, is used as a reduced encoding of the input.

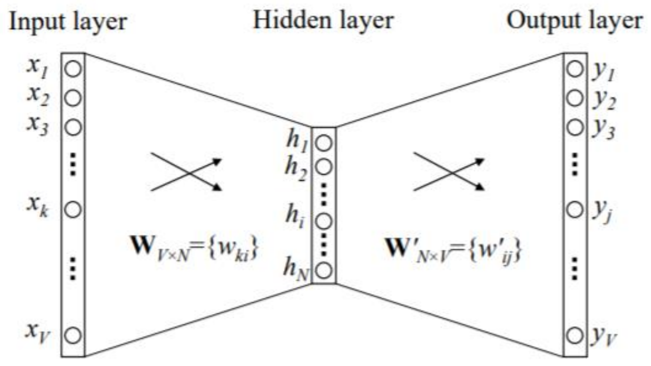


Figure 6.3.1 Autoencoder network with one hidden layer, from [96].

As an example, an autoencoder is applied to the one-hot vector example in Table 6.1. The autoencoder with a hidden layer of size 2 is trained and achieves zero loss. The reduced encoding is shown in the third column of Table 6.1. In this ideal example, these encodings can fully reconstruct the original one-hot vectors using the trained weights, therefore they can be used equivalently to one-hot vectors. Plotting the two values of the reduced encodings as x and y coordinates gives Figure 6.3.2. The locations of words are split apart evenly, indicating that they have little correlation with one another. Note that the result in this example is not deterministic, the network may converge on a different set of weights and encodings that can fully reconstruct the input.

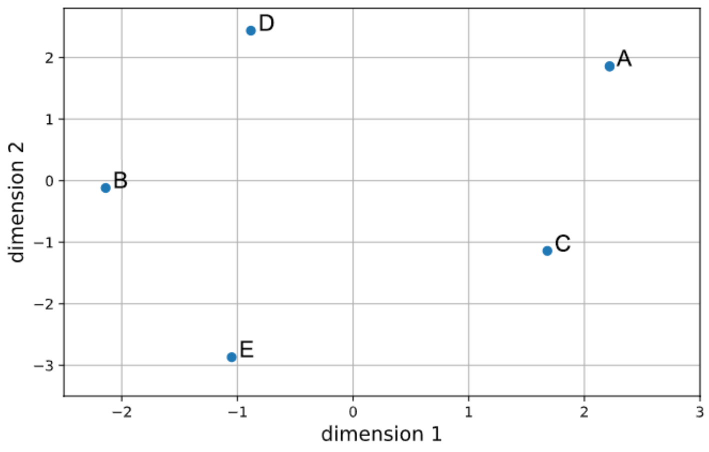


Figure 6.3.2 Reduced encodings obtained by the autoencoder.

6.3.2 Word Embedding

Embedding is an advanced type of encoding that has the semantic meaning of the symbolic tokens embedded into the vector. Word embedding addresses both the sparsity problem and the lack of correlation in the one-hot vector. The technique originates from Natural Language Processing (NLP) research investigating the numerical representation of words and texts, hence it is more commonly referred to as *word embedding*. In the text analysis research, the equivalency of a token and a sequence is a *word* and a *document*, respectively. The collection of words is called *vocabulary* and the whole text dataset is a *corpus*. The following description uses terminologies from NLP and log data interchangeably.

In the language modelling field, the semantic and syntactic meanings of a word can be and should be captured in multi-dimensional vectors. In human language, many words can be used interchangeably; swapping words with synonyms may make little change to the meaning of a sentence. Therefore, the encodings of synonyms should be similar, or visually, their locations should be close together on a multidimensional plot. Figure 6.3.3 shows the concept of embedding using a few common words based on their semantic meanings. Note that compared to Figure 6.3.2, words of the same category are close together. Words describing fluids, such as water, milk, and juice have similar embedding values, while words from different parts of a sentence are generally split apart. This example is only a simplified case with a few words represented by a vector with length 2. The actual language model could include thousands of words in a much higher dimensional space. Each dimension represents some association among the words. Different words could be close by in one dimension, while apart in another.

Software logs may not have the semantic complexity as human languages, but they still follow certain rules. Log tokens are dependent on one another and their correlations can be complicated. Learning the log token's embeddings could improve performance by introducing more information to the classification model.

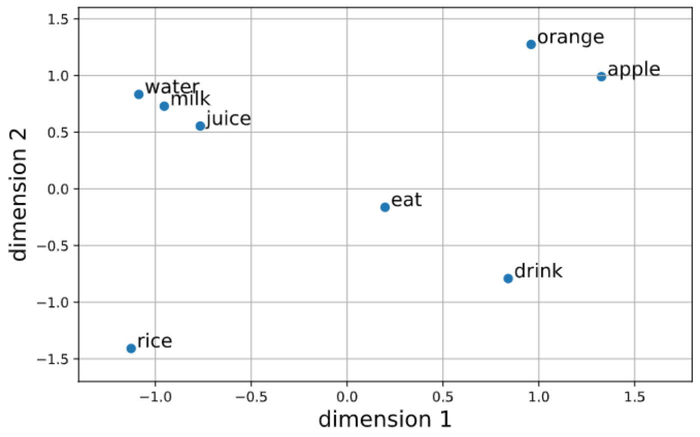


Figure 6.3.3 Word embeddings to capture semantic meanings

Finding the associations among words, or equivalently obtaining the word embeddings is one important task in language modelling. One intuitive method is to find how frequently two words occur together in a large corpus, which is obtained by iterating through all text sequences and accumulate co-occurrence counts of the vocabulary. If there are V words in the vocabulary, then the co-occurrence matrix would have a size of $V \times V$. Singular Value Decomposition (SVD) can then be used to reduce dimensionality and select the dominant singular vectors as embeddings. The SVD based methods are capable of capturing semantic and syntactic information effectively [97], but it lacks the flexibility of incorporating adjacent words with a distance of more than 1. Moreover, the low computational efficiency, including both memory and time, limits the use of this method.

6.3.3 Neural-Network-Based Embedding Methods

Neural-network-based methods or learning-based methods, such as the word2vec [98], achieves better results than SVD while addressing efficiency. The concept of word2vec is to use a neural network model – similar to an autoencoder – to predict context words given a centre word, or vice versa. Unlike the autoencoder, the word2vec approach no longer aims to reconstruct its input, but to predict the surrounding words instead.

In a simple setting, a word2vec model takes the form of an autoencoder. Training such a model uses the word pairs – two words occurring next to each other – from the dataset, one word called centre word u_c and the next one called outside word u_o. The one-hot encodings of u_c and u_o are used as input and target output for training, respectively. Since there can be many possible u_o for a word u_c within any proper document, the model would never predict precisely the next word. Instead, the trained model outputs a vector representing the most possible next word u_o and the probability scores $P(u_o|u_c)$.

Training such a model is similar to a multiclass regression model using a categorical cross-entropy loss function introduced in the previous chapter. The vector value of the middle layer of the trained model is the embedding representation of the input word.

In a more realistic approach, this model is extended to include multiple context words, including the words before and after u_c and with a distance of more than 1. When training the model, the target output is an average of the context words. This setting is called a *skip-gram* model. A skip-gram setting with four context words – two words before the centre word $u_{o,-1}, u_{o,-2}$ and two words after the centre word $u_{o,1}, u_{o,2}$ – is shown on the right of Figure 6.3.4. Alternatively, using the context words as input and the centre word as output is called the *continuous-bag-of-word* (CBOW) model, as shown on the left of Figure 6.3.4. In a CBOW model, the value of the middle layer is the embedding of the output word. Both models have far better computational efficiency than the SVD based methods and can achieve similar or better performance. The performance difference between skip-gram and CBOW is minimal. In practice, choosing between the two models depends on the amount and the importance of infrequent words.

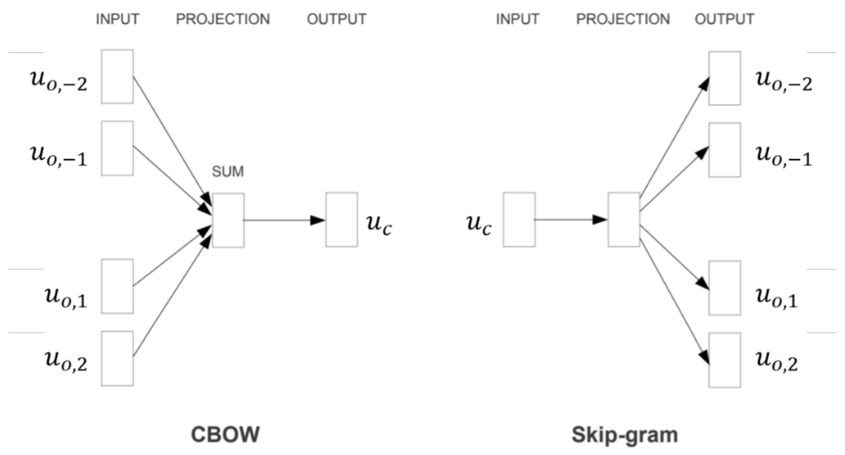


Figure 6.3.4 Neural network models to train word vectors. (from [98])

More advanced methods using deep neural networks with more layers, Global Vectors for Word Representation (GloVe) [99] as an example, can achieve better performance than word2vec in certain tasks, such as word analogy. However, in terms of system logs with a much smaller vocabulary, the word2vec solutions should be sufficient.

6.3.4 Implementation

In this study, a skip-gram model shown on the right of Figure 6.3.4 is trained using the HDFS dataset. It has an input size of 48, which equals the size of one-hot encoding and the number of unique log tokens. The size of the middle layer, or the embedding size $N = 16$, meaning that each token is represented by a vector with a length of 16. After training, only the first half of the model is needed, i.e. the mapping from the input token to the embedding vector.

Figure 6.3.5 visualizes the first two values of the embedding vectors of all 48 tokens in the HDFS dataset. To give a better visual presentation, the size of each bubble represents the occurrences of a log token. Specifically, it is proportional to $\log(\text{token\_count})$. In this graph, certain tokens cluster together and others split apart to some extend. This implies that the underlying correlations are being extracted. Most vector values being within (-1, 1) is another desirable result as this range is preferred for neural network training.

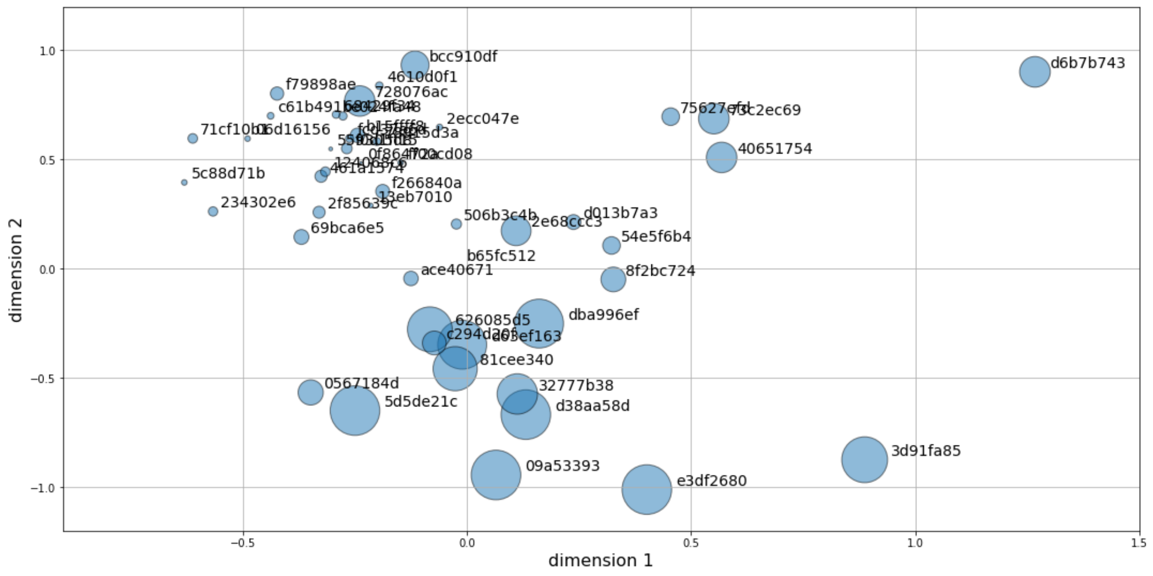


Figure 6.3.5 The first two dimensions of the log token embeddings.

After the embedding process, the numerical representations of log tokens are obtained in the form of an $N \times 1$ vector x_i. A token sequence is then represented as a vector sequence $x = [x_1, x_2, \dots, x_M]$ with a shape of $M \times N$ is created by stacking the embeddings of each token. The embedding process effectively creates a 2D matrix as shown by an example in Figure 6.3.6. This embedding matrix represents both sequential and sematic features of the token sequences and is ready to be processed by a classification model.

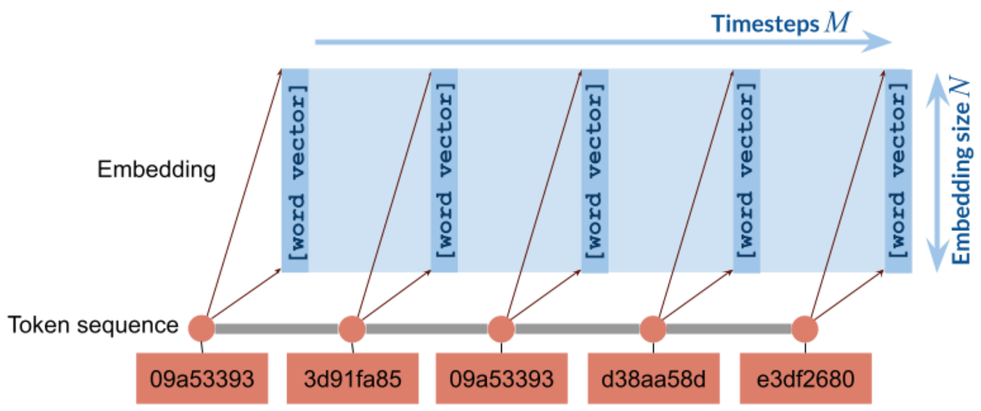


Figure 6.3.6 The embedding process applied to a token sequence.

6.4 Classification models

Deep learning models, or deep neural networks, are based on the concept of Artificial Neural Networks (ANN). While neural network research dates back to the 1940s [65], it is only the recent breakthroughs in computational capability that has unleashed some of its potentials. The recent development of various neural networks is mostly under the name of deep learning in order to differentiate from the older research. A traditional ANN, such as the Multi-Layer Perceptron (MLP) examined in the previous chapter, is less effective when dealing with data samples with large sizes or more than one dimension. Data in 2D matrices such as the one in this study, or 3D matrices such as a colour image, are difficult to fit the shape of an MLP's input. The recent development of neural networks addresses these issues and has quickly become the dominant approach for many problems.

Two deep learning structures are selected and examined in this study to perform classification tasks: 1) a Recurrent Neural Network (RNN) that processes a sequence of vectors in order [100] and, 2) a Convolutional Neural Network (CNN) that searches for 2D patterns from the vector sequence.

6.4.1 Recurrent Neural Networks

RNN applies recurrence to the neurons or layers in a neural network. The typical structure is based on the form of a layered neural network – an MLP – and the recurrence is applied to the intermediate layers. While an MLP is able to process an input vector x_1 of $N \times 1$, an RNN iteratively processes a sequence of such vectors $[x_1, x_2, \dots, x_M]$ with a shape of $M \times N$. It does so by retaining the hidden layers' values h and adding them back to the same layer for the next vector in the sequence. For example, the forward propagation of an MLP with 2 hidden layers using an input vector x_1 is represented as:

$$\begin{aligned} h_{1,1} &= S\left(W_{x,1}x_1 + b_1\right) \\ h_{2,1} &= S\left(W_{x,2}h_{1,1} + b_2\right) \\ \hat{y}_1 &= S\left(W_{x,3}h_{2,1} + b_3\right) \end{aligned} \tag{6.1}$$

where $h_{1,1}$, $h_{2,1}$, and $\hat{y}_1$ are the values of two hidden layers and model output, respectively. W's and b's are the weights and biases of the full connections. The first subscript refers to the number of layers and the second subscript refers to the sequential order of calculation, called the *timestep*. In the MLP case, only 1 calculation is performed, while RNN has M timesteps of calculations. The function S is the choice of the activation function.

Equation (6.1) also applies to the first iteration of an RNN, i.e., processing the first vector x_1 of the input. When processing the second input x_2 during the second timestep, the previous values of hidden layer h_1 and h_2 are added with another set of weights W_h:

$$\begin{aligned} h_{1,2} &= S\left(W_{x,1}x_2 + W_{h,1}h_{1,1} + b_1\right) \\ h_{2,2} &= S\left(W_{x,2}h_{1,2} + W_{h,2}h_{2,1} + b_2\right) \\ \hat{y}_2 &= S\left(W_{x,3}h_{2,2} + b_3\right) \end{aligned} \tag{6.2}$$

For a length-M vector sequence, the processing is executed M times until a final $\hat{y}$ is produced. The recurrence of hidden layers effectively acts as a memory, retaining the information from the previous timestep. In other words, the values from every timestep have a contribution to the final output.

This RNN is graphically presented in Figure 6.4.1. The iteration of each timestep is shown on the right side of the figure. The timesteps are unrolled in the horizontal direction, and for each timestep, the input x propagates through two hidden layers in the vertical direction. Since the calculation in each timestep is the same, a compact representation omitting the timesteps is often used as shown on the left. For classification, the output vector y is often processed by a full connection layer to further condense the output to the size of class labels.

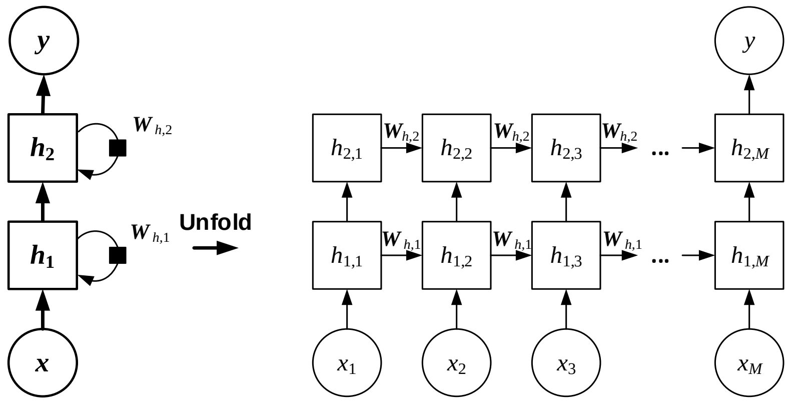


Figure 6.4.1 Structure of a stacked 2-layer RNN.

6.4.1.1 Variations of RNN Cell

RNN research commonly uses the term *cell* to refer to the hidden layers in Figure 6.4.1. The calculation of a typical RNN cell described in the previous subsection can be written as follows:

$$h_t = \tanh(W_x x_t + W_h h_{t-1} + b) \tag{6.3}$$

Compared to equation (6.1), this representation uses t and $t-1$ to represent the current and previous timesteps. The tanh activation is a common choice for most RNN cells.

RNN in this regular form has a critical issue called gradient vanishing when processing long sequences. The contribution of inputs of early timesteps are attenuated over the iteration and the gradients of these early timesteps become diminished when a sequence gets long. As a result, the training that relies on gradients would be ineffective. When the data sequence contains long-term dependencies, a regular RNN struggles to learn and fit the data [101]. Several RNN variations and training techniques are proposed to address this issue and the Long Short-Term Memory (LSTM) cell is one of the most successful.

The LSTM addresses this issue using a non-linear gating mechanism to regulate the information flow of the unit [102]. LSTM is built upon a regular RNN cell, containing a hidden state h_t that feeds back in the next timestep. Additionally, LSTM has a *cell state* parameter called C_t. For the current timestep t, the cell state *candidate* $\tilde{C}_t$ is first obtained

$$\tilde{C}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_C) \tag{6.4}$$

Note that the calculation is the same as equation (6.3). The term $W_x x_t + W_h h_{t-1}$ is combined and written as $W_c \cdot [h_{t-1}, x_t]$ for simplicity.

The LSTM then incorporates three gates that throttle the cell state and new input. These are called the forget gate f_t, input gate i_t, and output gate o_t. These gates have weights that are adaptive to the current input x_t and the previous hidden status h_{t-1}. A sigmoid function is applied to produce an output between 0 and 1:

$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right) \tag{6.5}$$

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{6.6}$$

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \tag{6.7}$$

The new cell state C_t is the summation of the previous cell sate C_{t-1} and the current candidate $\tilde{C}_t$, with the gate throttling:

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \tag{6.8}$$

where $\odot$ denotes element-wise multiplication of matrices, or Hadamard product.

The hidden state h_t is also throttled by the output gate:

$$h_t = o_t \odot \tanh C_t \tag{6.9}$$

Equations (6.4) - (6.9) describe the calculations within an LSTM cell. A graphical representation is shown in Figure 6.4.2. Because of the added gates and the cell state, an LSTM cell contains 3 times more parameters than a regular RNN. When stacking multiple LSTM layers as the structure in Figure 6.4.1, the hidden state h is passed to the next layer, while the cell state C is not. Therefore, from a high-level view, an LSTM has the same structure as a regular RNN like the one shown in Figure 6.4.1.

6.4.1.2 The RNN Classification Model

In this study, the fault detection for the HDFS system is a binary classification problem. The RNN model tailored for this problem is shown in Figure 6.4.3. The input log sequence is first converted into vector sequence using word embedding explained in Section 6.3. The result is an $M \times N$ embedding matrix, where M is the length or timesteps of the sequence and N is the number of embedding dimensions. The RNN layers then take the $M \times N$ matrix as input and output a vector $\hat{y}$. The output $\hat{y}$ is then processed by a full-connection neural network to condense to the final output classes of normal or faulty. The output layer has a softmax activation function to obtain a probability distribution among the two labels. The model can be adapted to a diagnosis setting by changing the final layer to the number of faults, without major alteration of the structure.

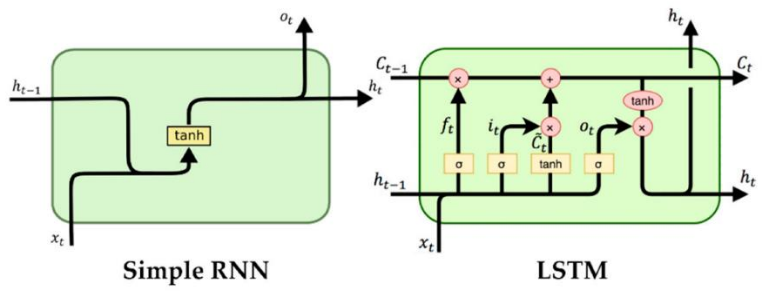


Figure 6.4.2 The calculations of a regular RNN and an LSTM cell.

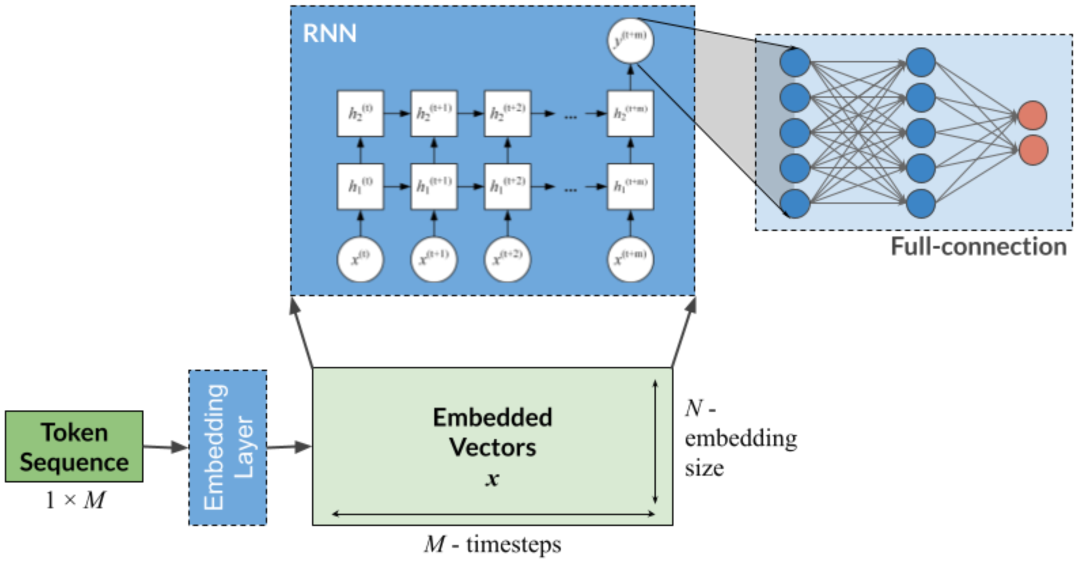


Figure 6.4.3 The RNN classification model structure.

6.4.2 Convolutional Neural Networks

Another approach to process high dimensional data is the Convolutional Neural Networks (CNN). While RNN was created to handle temporal behaviours in sequences, the CNN was originally designed to capture spatial features from image data. Since CNN's first successful application in recognizing hand-written digits [103], it quickly became the foundation of learning-based computer vision research. The trainable parameters in CNN are a set of *kernels* or filters, which are small matrices of a certain shape. In image classification tasks, the filters scan the entire image as a way of finding image features.

With some adaptation, the concept of convolution can be applied to sequential data as well, such as the embeddings of log tokens.

6.4.2.1 Convolution in Image Processing

The *matrix convolution* operation is commonly seen in image processing between a large matrix and a small one. The large matrix f often represents an image, and the small one w is a kernel or filter. The convolution $g = f * w$ is defined as:

$$g_{m,n} = (f * w)_{m,n} = \sum_{i}^{m} \sum_{j}^{n} w_{i,j} f_{m-j,n-k} \tag{6.10}$$

where the subscript m, n are indices of matrix elements.

In other words, the convolution applies Frobenius inner product – the sum of products of corresponding elements in two same-sized matrices – between the kernel w and a kernel-sized portion of the large matrix f. Effectively, the kernel scans the whole image produces a filtered version of the original one. Note that this definition is for matrix convolution in the image processing field; convolution has different definitions in other areas such as signal processing.

The upper part of Figure 6.4.4 illustrates the convolution of a 6×6 matrix f and a 3×3 kernel w. The highlighted portion indicates the first element of the convolved matrix g and the matrix elements for the calculation. If f represents a grayscale image – where large values represent dark colours and a zero represents white, as shown in the lower half of Figure 6.4.4 – then the filter is effectively a vertical edge detector. It highlights the contrasting portion of the original image, which is the vertical line in the middle. Many other formulations of kernels exist for various image processing purposes, such as edge detection, sharpening, and lowpass/blur.

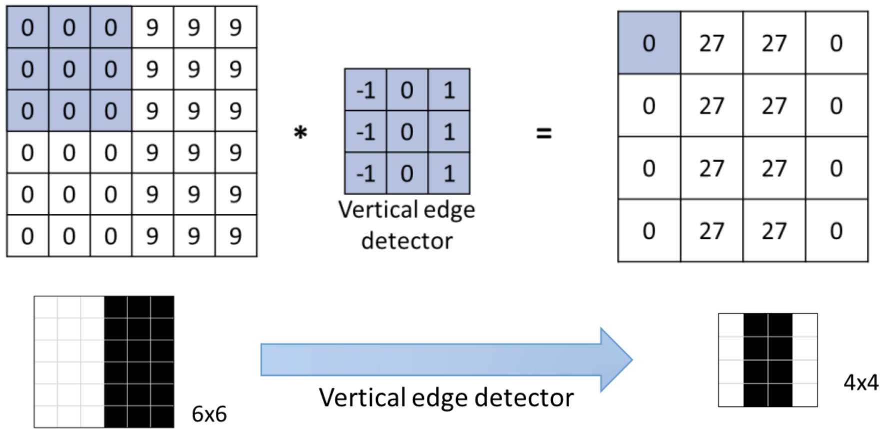


Figure 6.4.4 The convolution of a 6×6 matrix f and a 3×3 kernel w.

A CNN uses kernels as trainable parameters, meaning that the kernels are self-adapted to the training data in order to find the best configurations. An example of the CNN processing an infrared grayscale image is shown in Figure 6.4.5. Each convolutional layer consists of multiple same-sized kernels that convolve with the input matrix. The convolution generates a number of matrices of the same size, which are stacked and ready to be processed by another convolutional layer. A subsampling process, commonly known as a pooling layer, is often used after each convolution to reduce the size of the output matrix while keeping the significant values of the matrix. Multiple convolution-pooling pairs are used in sequence to create a multi-layer CNN. After a number of convolutions, the feature matrix is flattened into a vector – by placing all elements in a single column – which is then processed by a full connection network to reduce to class labels.

Apart from handling high-dimensional data directly, the two main advantages of a CNN over an MLP are sparse connectivity and the ability to capture spatial relations. Take the image processing as an example, each kernel in a convolutional layer applies to the entire image, but has only one set of trainable parameters. This means the kernel has a sparse, rather than a full connection to every pixel of the input image, reducing the number of parameters for training. More importantly, because each kernel applies to the entire image in a 2D manner, the spatial features of an interesting object can be captured regardless of their locations across the image. Concretely, if a CNN is trained to recognize a person, it will detect the targets even if they appear in a different location. A full connection network is unable to infer such changes as the connections are tied to each pixel.

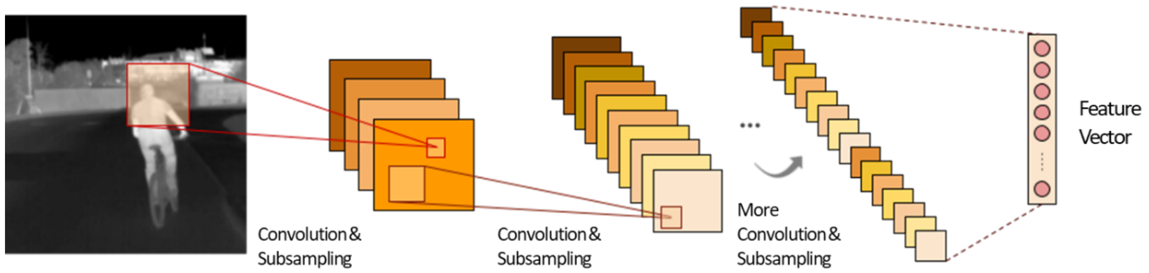


Figure 6.4.5 A Typical 2-D CNN architecture.

6.4.2.2 Sequential Convolution

In almost all computer vision applications, the kernels have a square shape and scan the image in both horizontal and vertical directions. This is because the image features to be captured are in two dimensions. However, for sequential data such as the log sequences in this study, the data only show temporal behaviour in one direction. For example, for the embedding matrix with a shape of $M \times N$ – with M timesteps and N embedding dimensions – a kernel that scans along the M timesteps would be sufficient. These kernels have a width of N same as the input matrix, and an arbitrary length smaller than the timesteps M. This setting is called one-dimensional (1D) convolution. A simple example with one 3×4 kernel convolving a $3 \times M$ sequence sample is shown in Figure 6.4.6. The kernel scans along the timestep direction and the convolution result is a vector. It is common to use multiple kernels, which result in multiple sequences. Then another layer of convolution can be applied. When constructing a neural network layer, stacking multiple convolution layers may help provide additional complexity to fit the data.

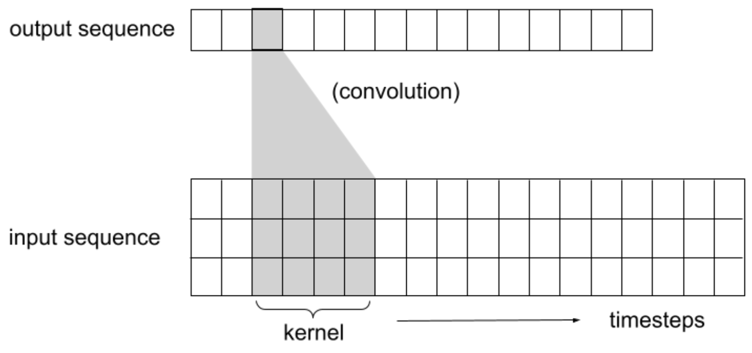


Figure 6.4.6 Convolution in 1D.

6.4.2.3 The CNN Classification Model

The classification model using CNN for the fault detection task is shown in Figure 6.4.7. The sequence sample first goes through an embedding process same as the one in the RNN model. A number of CNN layers process these embeddings and produce a feature vector. The feature vector is finally reduced to class labels via a fully connected network. The size and number of convolutional layers and full-connection layers can be configured to scale with the detection or diagnosis problem.

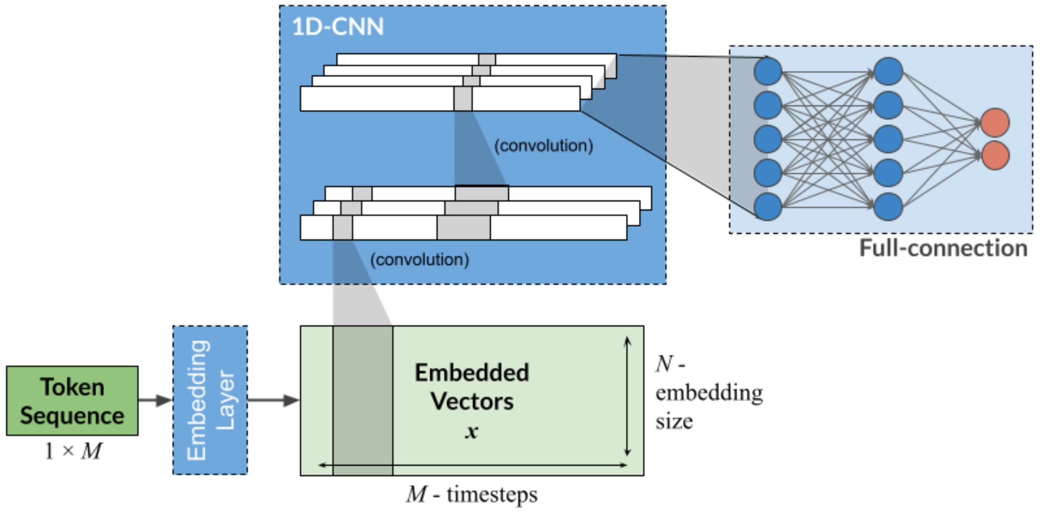


Figure 6.4.7 The 1D-CNN classification model structure.

6.5 Implementation

Both RNN and CNN models are implemented in Python using the TensorFlow 2.x package. The main platform for testing was the Google Colab – a cloud computing environment for executing Python programs – and a local machine running Windows 10. The hardware available from the Colab platform includes the Intel Xeon processor, Tesla P100 GPU, and 24GB memory. The local machine has an Intel i7 processor, an NVidia RTX 2080Ti GPU, and 32GB memory.

The vectorization model of log sequences are trained using the *gensim* package [104], a semantic modelling library that includes word2vec and other vectorization algorithms. The vectorization model produces embedding vectors of size 16. Training of the embeddings is separate from the detection model. Since the word2vec model is in a neural network form, the trained parameters are transferred to the detection model as the embedding layer as shown in Figure 6.4.3 and Figure 6.4.7.

6.5.1 Training Strategy for Imbalanced Dataset

As mentioned in Section 6.1.3, the HDFS dataset is highly imbalanced with significantly more normal samples than faulty ones. Training a model directly using such a dataset will lead to imbalanced performance within the two labels. Specifically, a trained model tends to “favour” the normal label if there are more normal samples in the training data. In an extreme case, a model simply deciding all samples as normal would achieve more than 97% overall accuracy, but it would be unusable for fault detection. Therefore, the model needs to be trained in a balanced way, and impartial metrics should be used in addition to the accuracy.

There are two general approaches to balance the data for training: training with class weighting and dataset oversampling. Class weighting applies a weight factor to the gradients based on the sample’s class, effectively changing the learning rate. For example, a sample of the minority class would have a higher weight, meaning that the network parameters are updated in a faster way, while a sample of the majority class has a smaller contribution to the training. Depending on how the gradient is used, applying class weighting may require the training algorithm to make adaptations. Therefore, it may not be a universal method for balancing datasets.

On the other hand, the oversampling technique reconstructs the training dataset instead of changing the training algorithm. The samples of the minority class are duplicated and added to the training set, such that the ratio between the two classes achieves 50:50. The new balanced dataset is then shuffled and used as a regular dataset for training. The alteration of the training set means that the training metrics no longer represent the actual performance. A different set of data with the correct class ratio, such as a validation set, is needed to monitor the training process. Nevertheless, the oversampling technique does not affect the training process and therefore a safer choice when experimenting with different optimization algorithms. It is therefore used in this study to train the model using the HDFS dataset.

Because of the imbalance, the accuracy metric is no longer sufficient to properly represent a model’s performance. Precision and recall scores provide a better description when evaluating individual classes. The F1 score which combines precision and recall is also used during the evaluation. These metrics are explained in more detail in Section 6.6.

6.5.2 Data Selection

As mentioned in Section 6.1.3, the sequence samples processed from the HDFS dataset have variable lengths as well as durations. These variations represent the difference in executing various tasks. Since the dataset was collected with an arbitrary start and end timing, some sequences may contain more than one task execution. The labels only reflect the status of a sample by the end of the data collection, but the time when a failure or fault is inflicted is unsure for these samples. As a result, long sequences may contain perfectly normal segments before exhibiting faulty behaviour. These normal patterns may potentially confuse the models as they are part of a faulty sample. Moreover, the deep learning models must have a definite input size, longer sequences have to be trimmed in order to fit the model’s input. This may induce further problems as the trimming risks cutting away faulty log patterns. Without extended knowledge of the HDFS system log, the safer option is to ignore the long sequences from the dataset.

Furthermore, the removal of long sequences is based on time duration, rather than length. This is from the practical consideration that automated fault detection systems are often based on the data collected from a certain period of time. In this study, a duration threshold $T_{\max}$ is set to select eligible samples for training and evaluation. A list of values varying from 2 minutes to 30 minutes is experimented to evaluate the models' performance on the sequence samples with different lengths. A larger $T_{\max}$ means more samples and longer ones are selected.

After data selection, the samples are split into training, validation, and testing sets with a ratio of 64%:16%:20%. The split is done with stratification, meaning that the class ratio is kept constant across all three sets.

6.5.3 Model Tuning

Tuning a deep learning model, or hyperparameter tuning, refers to the process of finding the best network configuration that produces the best performance. The hyperparameters of a neural network model include the number of layers, the size of each layer, the activation functions, and other layer configurations. The tuning process is carried out empirically through trial and error using the training set and validation set. The testing set is reserved and only shown to the model during the testing process.

The best network configurations for RNN and CNN are shown in Table 6.2 and Table 6.3, respectively. In each table, a forward pass is listed from top to bottom. The input of the models is the token sequence with a length of 250. Shorter samples are padded with a placeholder token in the front, and long samples are excluded in the training and evaluation process. An embedding layer with the pre-trained weights converts these token sequences into vector sequences, i.e., a 2D matrix, with a length of $M = 250$ and a width of $N = 16$. Next, an RNN model processes the embeddings with an LSTM layer, while the CNN model uses a convolution-pooling pair. The output of LSTM and convolution layers are both feed into full-connection layers to concentrate to class labels. The RNN model uses one full-connection layer, while the CNN model uses two. The final layer of both models is a single neuron with sigmoid activation. This is because the HDFS dataset is a detection problem, one activation value is enough to represent the likelihood of a fault. In the cause of a multiclass problem such as diagnosis, the final layer would contain a certain number of neurons with a softmax activation.

Table 6.2 The RNN model configuration.

Layer	Configuration	Shape
Input		(250)
Embedding	16 units	(250, 16)
LSTM layer	64 units, tanh activation	(64)
Full-connection	32 units, tanh activation	(32)
Output layer	1 unit, sigmoid activation	(1)
Total weights:	22,849 / 23,633	

Table 6.3 The CNN model configuration.

Layer	Configuration	Shape
Input		(250)
Embedding	16 units	(250, 16)
1D convolution	32 kernels, size=4, stride=1	(81, 32)
Global max pooling		(32)
Full-connection	32 units, ReLU activation	(32)
Full-connection	32 units, ReLU activation	(32)
Output layer	1 unit, sigmoid activation	(1)
Total weights:	6,273 / 7,057	

Although the structures of CNN and RNN models look similar, the detailed configurations are quite different. In fact, all layers are different other than the input, the output, and the embedding layers. A few interesting observations during the hyperparameter tuning process are:

1) In both models, one specialized layer – the LSTM layer or convolutional layer – is sufficient to obtain a decent result. This indicates that the dataset is relatively simple. Although the models are not particularly “deep”, they can be easily expanded with more layers and larger sizes to fit more complex datasets.
2) The RNN model prefers one full-connection layer before the output with long-tail activation functions such as tanh or sigmoid, while the CNN model produces better results with two full-connection layers with Rectified Linear Unit (ReLU) activations.
3) Global max pooling layers are used in the CNN model after convolution. Global max pooling is aggressive subsampling that represents a matrix by its largest element. It is different from image processing common practices that use regular max pooling, which is subsampling at a certain rate. This change results in a lower amount of weights, but produces better accuracy.
4) Advanced gradient-based training algorithms are experimented, such as the Adam [105] and RMSprop [106]. Both methods produce quick and smooth convergence. Adam tends to train slightly faster than RMSprop, but the difference is insignificant.

The training of both models is very fast on GPU-enabled hardware. The typical training time until convergence is less than 10 minutes for both models.

6.6 Evaluation

For binary classification tasks, common evaluation metrics include accuracy, precision, recall, and F1 score. The latter three are particularly useful when evaluating a model using an imbalanced dataset.

6.6.1 Evaluation Metrics

For a sample with a label of either 0 – normal or 1 – faulty, the model's detection result is defined as:

- True positive, the sample has a label of 1 and the model detects correctly.
- True negative, the sample has a label of 0 and the model detects correctly.
- False negative, the sample has a label of 1 and the model detects as class 0.
- False positive, the sample has a label of 0 and the model detects as class 1.

For a testing set, the number of true positives, true negatives, false positives, and false negatives are denoted as $n_{TP}, n_{TN}, n_{FP}, n_{FN}$, respectively. The four evaluation metrics are defined as follows:

- Accuracy: $\alpha = \frac{n_{TN}+n_{TP}}{n_{TN}+n_{FP}+n_{FN}+n_{TP}}$
- Precision: $p = \frac{n_{TP}}{n_{FP}+n_{TP}}$
- Recall: $r = \frac{n_{TP}}{n_{FN}+n_{TP}}$

- F1 score: $F_1 = \frac{2 \times p \times r}{p+r}$

While the accuracy indicates the overall performance, it shows little discrimination between the two classes. Precision and recall emphasize false positives and false negatives, respectively. F1 score is the combination of precision and recall, and therefore a better overall metric than the accuracy in an imbalanced dataset setting. All four metrics are demonstrated in this evaluation.

6.6.2 Evaluation Result

In this test, the data selection described in Section 6.5.2 uses a max duration $T_{\max}$ = 120s, giving a total of 101,245 samples for training and evaluation. Among them there are 6,422 faulty samples. An RNN and a CNN model are trained using the same training set. A testing set containing 20% of all samples is used to evaluate both models. The testing set has an imbalanced ratio between normal and faulty samples, the same as the training set. The probability threshold to determine whether a prediction is normal (0) or faulty (1) is 0.5, the default threshold. Table 6.4 shows the performance of both models at the default threshold. Note that the values in this table are not definitive as the model's final weights are subject to random initialization during the training process. Test results that are most reproducible are selected to display in this table.

Table 6.4 Model performance on the test dataset.

	Accuracy	**Precision**	**Recall**	**F1-score**
RNN	99.96%	99.46%	99.92%	99.69%
CNN	99.96%	99.46%	99.84%	99.65%

Based on Table 6.4, both models achieve very high accuracy of over 99%. There is no performance difference between CNN and RNN in terms of accuracy. When examining class-specific metrics, the RNN produces marginally better recall than the CNN. The 99.92% recall means that only one false negative is produced. As a result, the RNN's F1-score is slightly better than the CNN, however, the difference is insignificant. Note that the F1 score gives a different perspective from the accuracy and distinguishes between the two models.

6.6.3 Evaluation with Long Sequences

With the max duration $T_{\max}$ set to larger values including 240s, 480s, 960s, and 1920s, more data samples are selected for evaluation. A set of CNN and RNN models are trained using the selected samples. Their performance in terms of accuracy and F1 scores are shown in Table 6.5. With the increase of sequence duration, both models see a decrease in their performance. The decrease is less evident using the accuracy metrics, as both CNN and RNN show over 99% accuracy for all sequence durations. However, the decrease is clearly revealed with the F1 score. For the sequences that last up to 32 minutes (1920 seconds), the F1 score is only 95% and 91% for CNN and RNN, respectively.

Table 6.5 Model performance using longer sequence samples

T_{max}	Accuracy on testing set		F1 score on testing set	
	CNN	**RNN**	**CNN**	**RNN**
120s	99.96%	99.96%	99.65%	99.69%
240s	99.84%	99.74%	98.90%	97.80%
480s	99.84%	99.52%	98.59%	95.51%
960s	99.61%	99.39%	96.32%	93.64%
1920s	99.56%	99.32%	95.19%	91.69%

The difference between accuracy and F1 score is caused by the imbalance of the dataset. The accuracy metric treats both true positive and true negative equally, whereas precision and recall emphasize true positives. Because in this imbalanced dataset, positive samples – i.e. faulty samples – are scarce, false detections drag down the precision, recall, and F1 scores more quickly than they do to the accuracy. Therefore, the table reveals that the models are getting fewer true positives for longer sequences, which is worth further investigation as the true positives are more important than true negatives in a detection task.

6.7 Conclusion

This chapter takes a deep neural network approach towards the automated log analysis problem. This recently developed branch of machine learning is capable of processing matrices in two or more dimensions, enabling more complex feature engineering for the log data. Specifically, the word embedding method, one of the innovations in language modelling, is adapted for log data.

Two classification models, a CNN model using 1D convolution and an RNN model using LSTM cell are constructed, implemented, and fine-tuned. A training process with an emphasis on the imbalance of the dataset is completed. Both models show exceptional performance in terms of both accuracy and F1 score using the HDFS log samples up to a certain duration.

However, evaluation using longer samples reveals a decrease in performance for both models. The models produce fewer true positives and F1 scores when longer sequence samples are included in the training. It is clear that both models still have room for improvement. The next chapter explores new neural network structures to utilize the timestamp information from log data.

Chapter 7 Irregular Timestamp Integration

The methodologies presented in the previous chapters focus on the log events – represented by the log tokens – and a sequential combination of such elements. However, the timestamp as an important element of the log messages is unused. Most studies in the field have also opted to drop the log messages' timestamps. This may be attributed to the preconception that timestamps generally contain less important information than the log events, but no studies have shown such evidence. In fact, the timestamps in log data generally have irregular intervals, meaning that the time difference between a log message and the previous one varies depending on the task execution. These time intervals between log messages potentially reflect the health status of the software.

The utilization of timestamps can be difficult because of their irregular intervals. For the data mining and statistical feature methods, a lot of information is discarded – such as the sequential order – in the feature extraction process for efficiency purposes. Including timestamps information in these processes is difficult and rarely seen in existing studies. On the other hand, some of the deep learning research utilizing RNN models has included the timestamps when structuring their models [70], [71]. In these studies, the time elapsed between log messages, i.e., the delta time, forms a sequence that is processed by a separate RNN, in addition to the model that processes the token sequence. However, there are several issues with these studies. First, the reason to use timestamps is not discussed and the contribution of such information is unclear. Secondly, using a separate RNN classification model means that the time sequence is decoupled from the log token sequence, the timestamp pattern may be less meaningful without its correspondence to certain tokens. Lastly, the method of processing timestamps through delta time has only been investigated on RNN models and can be difficult to generalize to other deep learning structures, such as the CNN.

In this chapter, a novel method to incorporate irregular timestamps information in a unified model is proposed to address the aforementioned issues in the existing study. The models formulated using this approach are called *Timestamp-integrated models* (Ts models). Ts models treat the timestamped token sequences as digital signals in multi-dimensions – the first of its kind in the field – and apply interpolation to uniform the time intervals. The implementation of the interpolation takes the form of a neural network layer, so Ts models apply to both CNN and RNN structures.

This chapter is organized as follows. The systematic observation of timestamps showing their significance is presented in Section 7.1. The method to unify both timestamps and token sequences is introduced in Section 7.2. The method implementation is detailed in Section 7.3. Section 7.4 evaluates the proposed models and compares them with the base neural network models in the previous chapter.

7.1 Data Observations with Timestamps

The observations are based on the HDFS dataset previously introduced in Section 6.1. The following subsections examine the data from the statistical and microscopic perspectives.

7.1.1 Statistical Overview of the Log Sequences

As mentioned in the previous chapter, the HDFS dataset contains log sequence samples consisting of 48 unique tokens. A majority of the sequence samples have lengths between 10 to 40, with an average of 19.43. The distribution of sequence sample lengths does not show a clear pattern as shown in Figure 7.1.1 (reusing Figure 6.1.7). There is little distributive difference between the faulty and normal samples as shown in Figure 7.1.2 (reusing Figure 6.1.8), other than the short samples with a length less than 10 most likely being faulty.

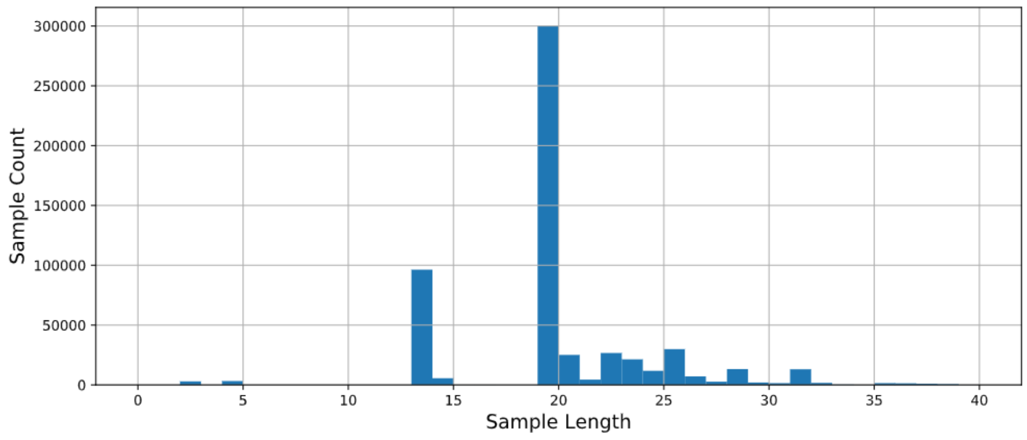


Figure 7.1.1 Histogram of sequence lengths (full dataset).

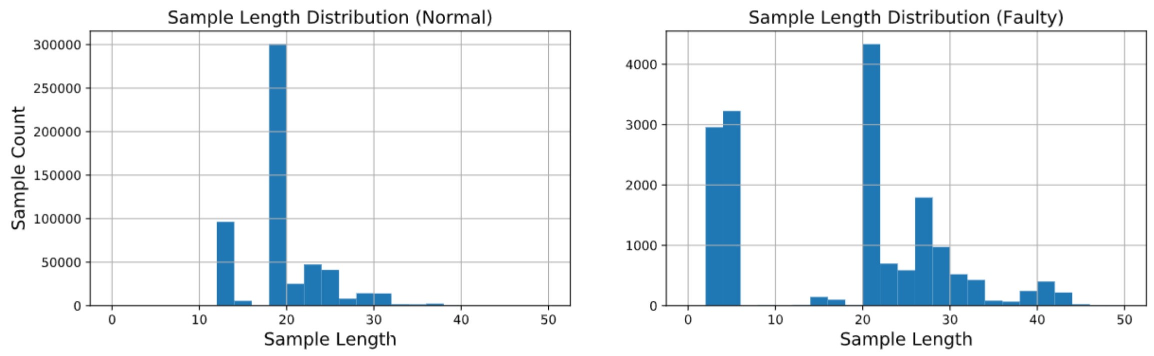


Figure 7.1.2 Histogram of sequence lengths by labels.

However, if the timestamps are taken into consideration, the distributions show a vastly different picture. The duration of each sample ranges from 0.5 to 54,000 seconds, where a majority of them lie within 120 seconds. A histogram of all sequence durations, is shown in Figure 7.1.3. The distribution follows a clear bimodal pattern in the 0 – 120s range as shown on the top graph. A majority of the samples follow a Gaussian distribution with a mean of 40 seconds. A smaller amount of short-lived sequences clutter around the very left of the graph. The contrast with the length distribution indicates that the execution time reflects the dynamics of software tasks that may not be captured using the event sequences only.

Noticeably there are quite a number of samples having longer durations up to 54,000 seconds as shown at the bottom of Figure 7.1.3. These samples are distributed randomly and clearly not outliers. Since the sequence samples are associated with block identifiers in the HDFS system, these longer samples are likely caused by the “busy” blocks getting multiple requests over the data collection period. In other words, they are possibly multiple sequences being joined together.

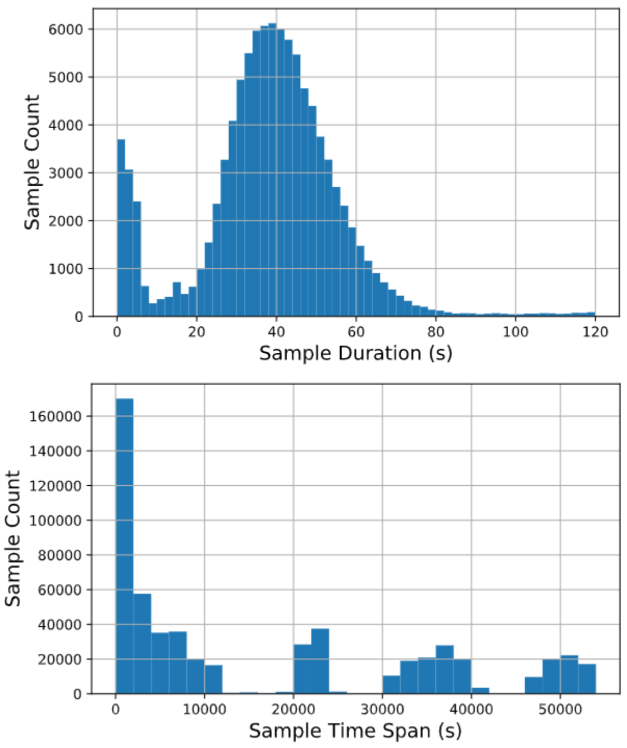


Figure 7.1.3 Histogram of sequence duration (full dataset).

Figure 7.1.4 shows the sequence durations histograms of normal and faulty samples. An interesting contrast can be observed compared to Figure 7.1.2. The normal data appears to be two Gaussian distributions with means at about 3s and 40s. In contrast, most faulty samples are short-lived and cluster within the 0 to 10 seconds range. There is a similar amount of faulty and normal samples within the 10 seconds limit. Longer faulty samples with a duration of more than 10 seconds do exist, but they are only a few and are barely distinguishable on the histogram.

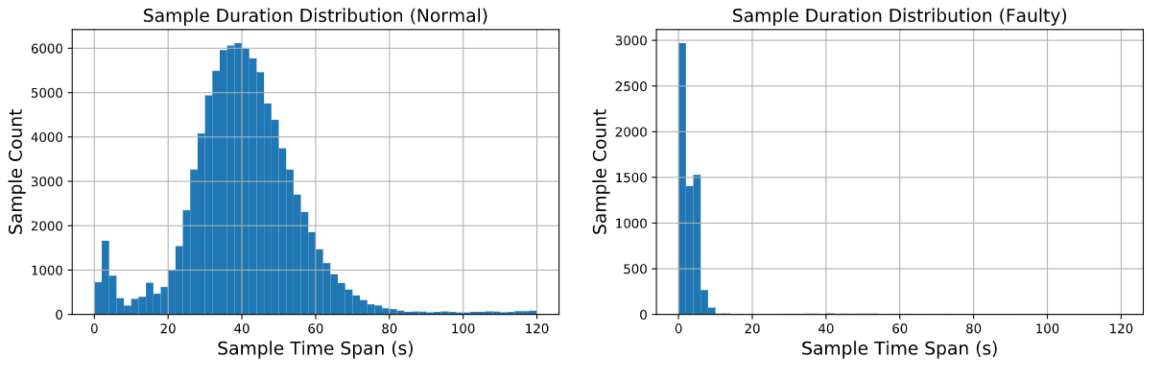


Figure 7.1.4 Histogram of sequence durations by labels.

As explained in the previous chapter, the sequential neural network models require a pre-defined input size that cannot be arbitrarily long. A general approach is to trim the longer sequence to the model's input, but it is not suitable for this study. This is because the timing of a fault inflicted is unknown, trimming the sequence risks cutting away faulty patterns. Therefore, the long samples are disregarded. A threshold $T_{\max}$ is set to select samples for training and evaluation. The effect of varying $T_{\max}$ is discussed in Section 6.6 as well as the rest of this chapter.

7.1.2 Microscopic observation

From a microscopic view, the difference between a sequence with and without timestamps is also significant. Figure 7.1.5 picks a normal sample with 12 data points, and plots it both even spaced (left) and by timestamps (right). Note that the absolute values of data points are meaningless, as they are categorical tokens created from the log messages. Because the timestamps have irregular intervals, some data points are closer together and some are separated away on the timestamp plot. For example, the timestamp plot clearly shows that there is a long span between the third and the fourth data points, which is not captured by the left plot. The appearance that some specific data points clustering or separating away may contain valuable information that contributes to the behaviour of a fault.

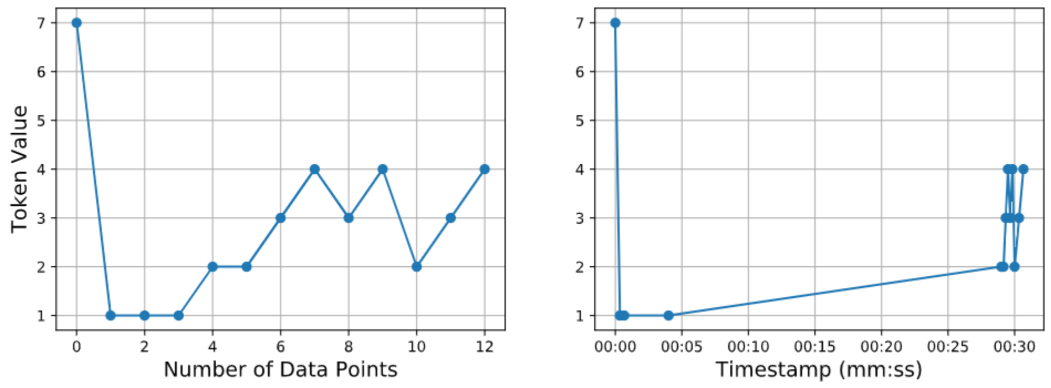


Figure 7.1.5 Sequence token plot of a healthy sample

As a summary of this section, the timestamps of a log sequence contain extra information in addition to the token sequence. This additional information can be revealed through both statistical and microscopical observation of the data. The difference between normal and faulty samples' histograms further shows that the conditions and the durations are correlated, indicating that the time information may potentially be utilized to improve a fault diagnosis system.

7.2 The Detection and Diagnosis Framework with Timestamps

To utilize the irregular timestamps along with the log sequences, the structure of deep learning models needs to be modified. One way of including this information in existing studies is to use an RNN to process the time difference between neighbouring token sequences, i.e., the delta time. Studies [70], [71] used the delta time to create a numerical sequence, which is handled in an RNN model parallel with the token processing model. While this method may be effective – the related study did not mention the contribution of this additional RNN – the time sequence itself is less meaningful without correlating to the token sequence. This section gives an overview of the proposed Ts models, which handle the irregular timestamps by resampling the token sequence. Timestamp processing is integrated with token processing in the one unified Ts model without adding a separate one.

7.2.1 Resampling as A Layer

Resampling is often applied to achieve time interval uniformity of a numerical sequence. The difficulty of applying resampling to the log sequence is evident: the tokens are categorical and cannot be used directly for calculations. Therefore, common resampling techniques requiring computation between neighbouring data points do not apply. Fortunately, since the deep learning models introduced in the previous chapter commonly include the embedding process that represents tokens as numerical vectors, resampling can be carried out at the embeddings level. Specifically, as an example shown in Figure 7.2.1, a token sequence with a length of $M = 5$ is first converted into an $M \times N$ embedding matrix, where N is the embedding size. The elapsed time between the column vectors are represented as gray spaces on the graph. Due to the nature of log data, these time intervals between the neighbouring column vectors are nonuniform. If treating the embedding sequence as a signal, an up-sampling process can be applied to each row of the embedding sequence. The resampling produces a vector sequence with a shape of $K \times N$ $(K > M)$, where the time intervals between the neighbouring column vectors are equal.

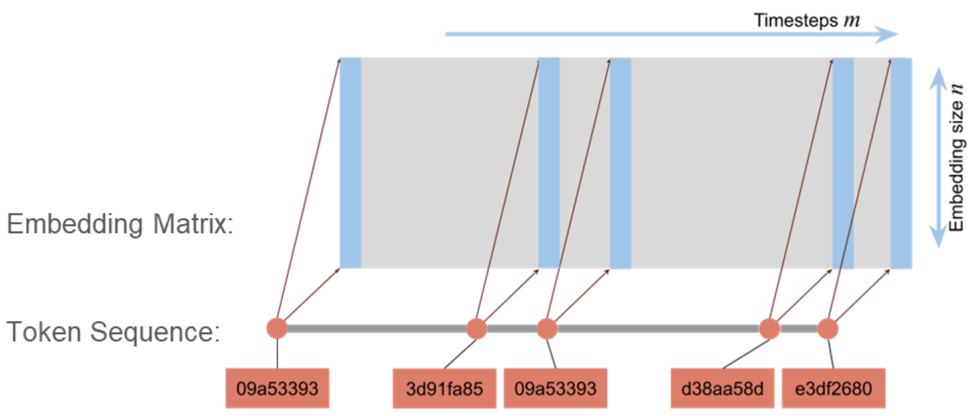


Figure 7.2.1 Resampling applied at the embeddings level.

While resampling a token sequence appears irrational, resampling the embedding vector sequence has its physical meaning. Since embedding is a process that represents tokens as points in a high dimensional space, the token sequence can be viewed as a signal consisting of these data points along the time dimension. Resampling these high dimensional signals and creating new data points in between the token points is therefore a reasonable process. An overview of the resample process is presented in Section 7.2.2 and detailed explanation is provided in Section 7.3.

7.2.2 Neural Network Model with Resample Layer

The resampling introduced in the previous subsection needs to be performed at the embeddings level, so it is preferably implemented as a layer in the neural network model. The structure of proposed Ts models is shown in Figure 7.2.2. Compared to the RNN model (Figure 6.4.3) and the CNN model (Figure 6.4.7) in the previous chapter, the main addition is a resample layer between the embedding process and the sequential classification process. This layer performs resampling – specifically, interpolation and extrapolation as introduced later in Section 7.3 – of the embedding vectors. As a result, the timestamps are required as input of the model.

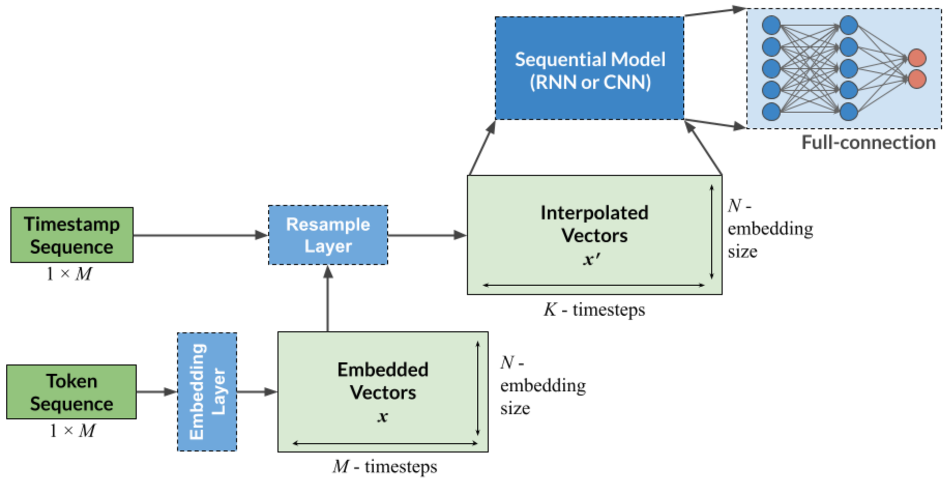


Figure 7.2.2 The overall model structure that integrates timestamps.

Figure 7.2.3 illustrates an example of the embedding and resample process under the new structure. The token sequence and its timestamps are taken as separate inputs. Apparently for each sample, the number of tokens equals the number of timestamps. They are aligned and represented in a vector form with a size of $1 \times M$. The token sequence is non-numerical, so it is first converted into an embedding matrix. Since the embedding process does not alter the columns, the embedding vectors with a size of $N \times M$ are still aligned with the timestamps, which have a size of $1 \times M$. Each row of the embedding can then be treated as a numerical signal and, with the timestamps provided, be processed with interpolation methods. The purpose of this interpolation is to take the nonuniform time intervals into consideration such that the output sequence (signal) reflects a correct temporal behaviour. The same interpolation process applies to every row of the embedding matrix, and the output of the resample process is a vector with a size $N \times K$, where K is determined by configurable parameters. Preferably K should be larger than M to ensure no data points are lost during the process; hence the process being up-sampling. A K smaller than M represents down-sampling, which also achieves time-uniformity.

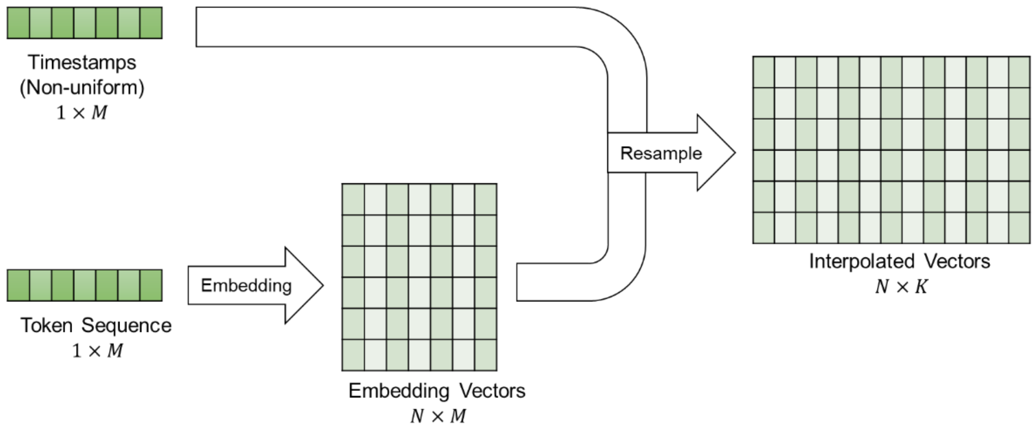


Figure 7.2.3 The embedding and resample process.

After resampling, the input samples with nonuniform intervals would have the same resolution and duration, so the timestamps are safely discarded. The interpolated vectors then proceed to an RNN or CNN model for classification, similar to the base models in Figure 6.4.3 and Figure 6.4.7 from the previous chapter. The output structure has the same configuration as the base models as well. It is clear that the addition of timestamp processing only changes the length of the embedding matrix and does not alter the classification part of the model. As a result, the method is universal to any network variations that process inputs in a sequential manner, such as the CNN and RNN to be introduced later in this chapter.

7.2.3 Fault Detection and Diagnosis Framework

The overall structure of the fault detection and diagnosis system integrating timestamps is shown in Figure 7.2.4. Similar to the one presented in the previous chapter, it consists of a training phase shown on the top of the graph and a diagnosis phase on the bottom. The log data goes through a parsing process to obtain their token sequences as well as timestamps. The deep learning model refers to the one proposed in Figure 7.2.2, which includes a resample layer to handle timestamps. Once the model is trained on labelled data, it can be deployed to perform detection or diagnosis tasks.

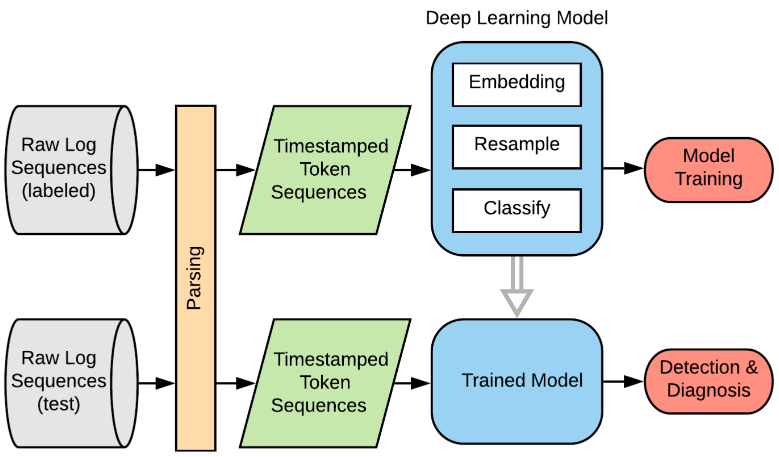


Figure 7.2.4 The Overview of Deep Learning Detection and Diagnosis System

7.3 Resample Methods

This section details the methods and implementation of resampling. As previously introduced, the resample process is preferably up-sampling since the intention is to extract finer details from the input. This means new data points are created between and after the existing embedding sequences. In other words, both interpolation and extrapolation are needed. Interpolation is more important as it reconstructs the dynamics in between existing data points, while extrapolation in this case is merely padding the sequence such that the output size is unified.

7.3.1 Interpolation

Several interpolation methods are introduced as follows. For the signals in one dimension, an input time series is denoted as (t, x):

$$\begin{aligned} t[M] &= [t_0, t_1, \dots, t_{M-1}] \\ x[M] &= [x_0, x_1, \dots, x_{M-1}] \end{aligned} \tag{7.1}$$

where M is the length of the sequence. An interpolation solves for a new series (t', x'):

$$\begin{aligned} t'[K] &= [t'_0, t'_1, \dots, t'_{K-1}] \\ x'[K] &= [x'_0, x'_1, \dots, x'_{K-1}] \end{aligned} \tag{7.2}$$

where K is the length of the interpolated sequence and larger than M.

The original sequence tuple (t, x) generally has irregular intervals. The output tuple (t', x') has a fixed interval T_s, a.k.a. the sampling rate. The interpolation is achieved by solving a function $P(t)$ that goes through all input data points (t_i, x_i). There are several interpolation methods for such numeric sequences [107]:

1. Zero-Order Hold (ZOH), or piecewise-constant interpolation, is the simplest and fastest way to even out the timestamps. It takes the closest sequence data point as the output. The interpolation result is a discrete function expressed as follows:

$$P(t) = x_i, \qquad \text{if } \ t_i < t < t_{i+1} \tag{7.3}$$

2. Linear interpolation is a local interpolation method that calculates new sample points by fitting a linear function using the adjacent two data points. It produces a continuous function, although its first-order derivative is still discrete at the original data points:

$$P(t) = \frac{x_{i+1} - x_i}{t_{i+1} - t_i} t + \frac{x_i t_{i+1} - x_{i+1} t_i}{t_{i+1} - t_i}, \qquad \text{if } \; t_i < t < t_{i+1} \tag{7.4}$$

3. For improved smoothness, cubic spline interpolation is another local interpolation option that fits a series of third-order polynomials (splines) $s_i(t)$ using four adjacent data points, while ensuring the whole function has continuous second-order derivative at all input data points:

$$P(t) = s_i(t), \qquad \text{if } \; t_i < t < t_{i+1} \tag{7.5}$$

where the third order polynomial $s_i(t) = a_i + b_i t + c_i t^2 + d_i t^3$ satisfies:

$$\begin{aligned} s_i(t_{i-1}) &= x_{i-1} \\ s_i(t_i) &= x_i \\ s_i'(t_i) &= s_{i+1}'(t_i) \\ s_i''(t_i) &= s_{i+1}''(t_i) \end{aligned} \tag{7.6}$$

4. Lagrange interpolation achieves continuity for all orders of derivatives by fitting a polynomial function with an order of $M - 1$. One way to obtain the polynomial function is as follows:

$$P(t) = \sum_{i=0}^{M-1} x_i L_i(t) \tag{7.7}$$

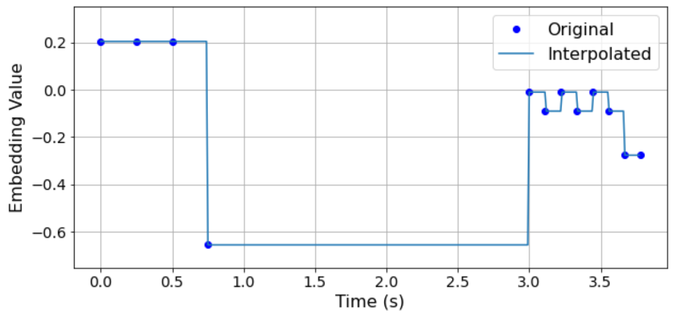
where $L_i(t)$ are the Lagrange basis polynomials and obtained as follows:

$$L_i(t) = \prod_{j=0, j\neq i}^{M-1} \frac{t - t_i}{t_i - t_j} \tag{7.8}$$

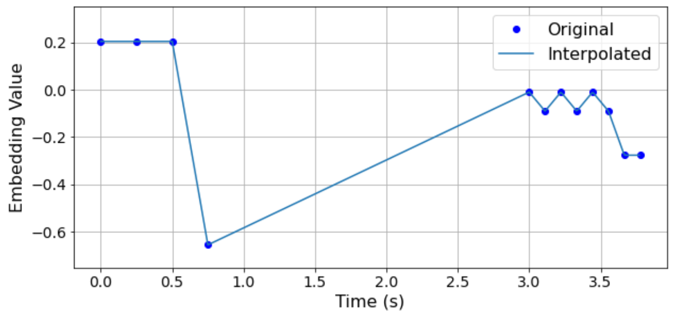
After obtaining the interpolated function $P(t)$, the resampled sequence $x'[K]$ is calculated by:

$$x_i' = P(t_i'), \quad i = 0,1, \dots, K-1 \tag{7.9}$$

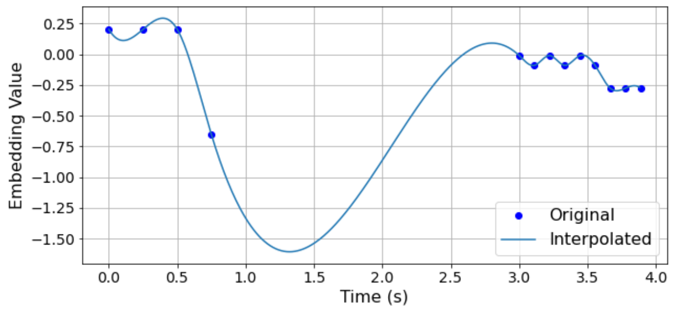
The set of figures in Figure 7.3.1 show the examples of interpolating a short sequence from the dataset. These examples represent one dimension (row) of the embedding matrix. Figures (a) through (d) correspond to the results of using ZOH, linear, cubic spline and Lagrange methods, respectively. The ZOH interpolation in Figure 7.3.1a produces a discrete function that contains mainly staircase patterns. Linear interpolation in Figure 7.3.1b gives a continuous function, showing improved smoothness over the ZOH method. Visually this reveals more varied features than Figure 7.3.1a. The cubic spline interpolation in Figure 7.3.1c appears very smooth, and the values of the splines are within a reasonable range. The Lagrange interpolation in Figure 7.3.1d shows similar smoothness as Figure 7.3.1c, but does not seem to bring more distinct features. Meanwhile, the Lagrange method creates unexpectedly large values – the range of y-axis is exceptionally large – which is likely unreasonable from a practical point of view.



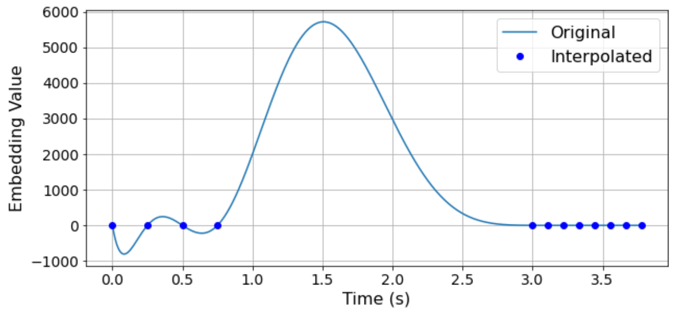
(a) Example of the ZOH interpolation.



(b) Example of the linear interpolation.



(c) Example of the cubic spline interpolation.



(d) Example of the Lagrange interpolation.

Figure 7.3.1 Examples of different interpolation methods.

Among the four methods, the ZOH and linear interpolation methods are selected for implementation and evaluation. ZOH is the most practical to implement and adds little computation overhead to the existing process. The linear method provides better smoothness and more distinct patterns than the ZOH method and appears to be a balance between feature richness and practicality. The Lagrange method is not preferred as its unexpectedly large values may cause unstable neural network training. The cubic spline interpolation appears to be a good candidate, but the computational requirement makes it less practical. The traditional method of calculating cubic splines of a size-M sequence requires solving an $M \times M$ tridiagonal linear system. Even with efficient algorithms [108], the cubic spline is still significantly more complex than the linear method. Since the computational requirement of the interpolation process can be quite demanding as discussed in Section 7.4, the cubic spline method is not chosen.

7.3.2 Extrapolation

The output of the resample layer is required to be a fixed size K, which is determined by the predefined sampling rate T_s and duration D_s. Since the input timestamp sequences have varied durations, the interpolation result using T_s may not match the fixed size K. Therefore, extrapolation is needed to extend the output sequence to the desired size by adding new data points at the end of the sequence. Extrapolation uses the same methods as interpolation and the feasible choices are ZOH and linear. Between the two, ZOH should be preferred rather than linear extrapolation, since the latter may lead to very large end values if the last two data points form a large slope.

An example of applying the described resampling – linear interpolation and ZOH extrapolation – to a timestamped sequence is shown in Figure 7.3.2. The original embedding value sequence with timestamps is shown on the left, and the resample result is on the right. A small enough T_s is chosen to capture the change of values, but not too small to increase unnecessary computational overhead. After resampling, all data points have equal intervals and all sequences use the same T_s, so the timestamp sequence can be safely discarded.

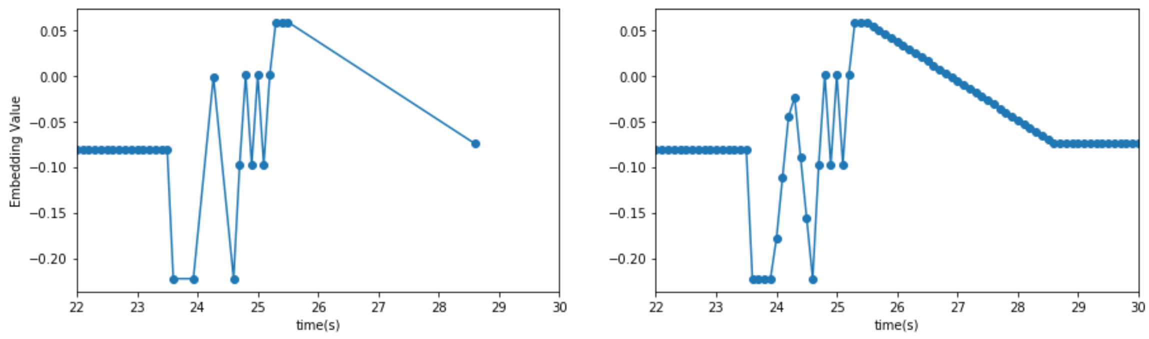


Figure 7.3.2 Example of linear interpolation with ZOH extrapolation.

7.3.3 Resample Implementation

The resample layer needs to be implemented in a batch form in order for it to be usable in neural network training. Each embedding matrix contains N sequences to be interpolated; processing them one by one takes too much time – so much that the network training would be unrealistic. The solution is to utilize parallel computing, which performs calculations for the whole embedding matrix at once. Practically, this means implementing the interpolation and extrapolation through matrix calculations.

Interpolation in the matrix form uses the same equations as single sequences presented in Section 7.3.1. The key is to prepare the variables as batches and avoid *for loops* in the code implementation. Figure 7.3.3 shows the pseudo code for linear interpolation function. This function takes N timestamped sequences as input and produces N interpolated sequences as output. Referring to equation (7.4), steps 1 to 3 prepares $t_i, t_{i+1}, x_i, x_{i+1}$for the calculation. Note that they are all in the form of $N \times K$ matrix, therefore having different names ts_{lo}, ts_{hi}, seq_{lo}, seq_{hi}. Steps 4 and 5 carry out equation (7.4) using matrix addition and multiplication.

By the time of writing this dissertation, none of the common deep learning libraries (TensorFlow or PyTorch) does interpolations described in this chapter, so the resample layer is implemented for the first time. Specifically, the resample layer is programmed as a standard Keras layer class, and the interpolations are coded using TensorFlow's Application Programming Interface (API). Packaging as a Keras class ensures the resample layer can be used seamlessly with many other layer classes provided by TensorFlow/Keras, while conforming matrix calculations with TensorFlow API automatically enables parallel computing. GPU acceleration can also be enabled when one is available. The link for the code implementation is provided in Appendix B.

```
Algorithm 1: Linear interpolation in matrix form
function ($ts$, $seq$):
Input: $N$ timestamped sequences with length $M$. $ts$ is an $N \times M$ matrix of time in
        seconds, $seq$ is an $N \times M$ matrix of embedding values.
Output: the interpolated sequence $seqInterp$ if size $N \times K$.
1. Create equal-interval reference timestamps $tsRef$ using the predefined duration
   and resample rate. $K$ denotes the length of $tsRef$.
2. Find $idx$, the indices of $tsRef$ values with regards to $ts$. $idx$ is an $N \times K$ matrix.
3. Gather the values of $ts$ and $seq$ at given index $idx$:
   $ts_{lo}$ = $ts$ values at $idx$
   $ts_{hi}$ = $ts$ values at $idx + 1$
   $seq_{lo}$ = $seq$ values at $idx$
   $seq_{lo}$ = $seq$ values at $idx + 1$
   $ts_{lo}$, $ts_{hi}$, $seq_{lo}$, $seq_{hi}$ are $N \times K$ matices.
4. Calculate the slope of linear interpolation: $slope = \frac{seq_{hi} - seq_{lo}}{ts_{hi} - ts_{lo}}$ (equation 7.4)
5. Calculate interpolated values: $seqInterp = seq_{lo} + (tsRef - ts_{lo}) \times slope$
   (equation 7.4).
```

Figure 7.3.3 Pseudo code for linear interpolation in matrix form.

7.3.4 Discussion on Interpolating Embeddings

The underlying assumption for applying interpolations is that there exists an original continuous signal, from which the data points are discretely observed. In this study, the software execution is treated as the continuous process, and the log data are discrete observations. Deducing the optimal interpolation method would require analyzing the software process as signals, but few studies have investigated this topic and it is beyond the scope of this research. Therefore, this study takes an experimental approach. A series of tests are designed in Section 7.5 to compare the performance of different methods and determine the best one for the classification task.

7.4 Implementation

The resample process is implemented in Python and packaged as a neural network layer that is compatible with the TensorFlow framework (refer to Appendix B for source code). Two parameters are pre-defined: the resample rate T_s and the sample duration D_s. The choice of T_s is preferably smaller than the shortest timestamp interval in the given sequence dataset, in the HDFS dataset it is set at 0.1s. Sample duration D_s is set to be the same as the longest sequence sample in the dataset, such as 120s.

Based on the proposed framework in Figure 7.2.2, two deep learning models are presented: the Ts-RNN and Ts-CNN. They are based on the RNN and CNN presented in the previous chapter. For each model, an ZOH and a linear variation are implemented. The cubic spline and Lagrange methods are not considered for the reasons explained in Section 7.3.1. As a result, there are in total six models to be evaluated:

1. CNN model same as in Chapter 6,
2. RNN model same as in Chapter 6,
3. Ts-RNN-ZOH, timestamp RNN model with ZOH interpolation,
4. Ts-CNN-ZOH, timestamp CNN model with ZOH interpolation,
5. Ts-RNN-Lin, timestamp RNN model with linear interpolation,
6. Ts-CNN-Lin, timestamp CNN model with linear interpolation.

The models' configurations are shown in Table 7.1 and Table 7.2, where the tables on the left show RNN and CNN base models copied from Table 6.2 and Table 6.3 in the previous chapter. The tables on the right show Ts models based on RNN and CNN.

Compared to the base models, the Ts models take an additional timestamp input ts and add a resample layer after embedding. The resample layer does not contain trainable parameters, so the total weights for the base model and the Ts model are the same.

Table 7.1 The model configurations using RNN.

RNN

Layer	**Configuration**	**Shape**
Input		(250)
Embedding	16 units	(250, 16)
LSTM layer	64 units, tanh activation	(64)
Full-connection	32 units, tanh activation	(32)
Output layer	1 unit, sigmoid activation	(1)
Total weights:	22,849 / 23,633	

Ts-RNN

Layer	**Configuration**	**Shape** $\boldsymbol{x}, \boldsymbol{ts}$
Input		(250), (250)
Embedding	16 units	(250, 16), (250)
Resample	T_s = 0.1s	(1201, 16)
LSTM layer	64 units, tanh activation	(64)
Full-connection	32 units, tanh activation	(32)
Output layer	1 unit, sigmoid activation	(1)
Total weights:	22,849 / 23,633	

Table 7.2 The model configurations using CNN.

CNN

Layer	**Configuration**	**Shape**
Input		(250)
Embedding	16 units	(250, 16)
1D convolution	32 kernels, size=4, stride=1	(81, 32)
Global max pooling		(32)
Full-connection	32 units, ReLU activation	(32)
Full-connection	32 units, ReLU activation	(32)
Output layer	1 unit, sigmoid activation	(1)
Total weights:	6,273 / 7,057	

Ts-CNN

Layer	**Configuration**	**Shape** $\boldsymbol{x}, \boldsymbol{ts}$
Input		(250), (250)
Embedding	16 units	(250, 16), (250)
Resample	T_s = 0.1s	(1201, 16)
1D convolution	32 kernels, size=8, stride=3	(332, 32)
Global max pooling		(32)
Full-connection	32 units, ReLU activation	(32)
Full-connection	32 units, ReLU activation	(32)
Output layer	1 unit, sigmoid activation	(1)
Total weights:	6,273 / 7,057	

All six models are trained and evaluated on GPU-enabled hardware platforms – a windows machine and the Google Colab cloud computing environment. Training Ts models takes longer than the base models, which is expected as the interpolation is a rather heavy process. Table 7.3 shows such comparisons using a portion of the dataset as an example. The Ts-CNN-Lin requires 5 times more time per epoch than the regular models, while training Ts-RNN-Lin requires more than twice the time of training a Ts-CNN-Lin. The processing time of Ts models is also directly related to the resample configuration – a longer duration leads to more computation time and memory usage. This is manageable when carrying out the tests in this chapter's evaluation section, but may run into memory shortage when the network is configured to process very long samples.

Table 7.3 Training comparison using a portion of the dataset.

	Optimizer	**Time per epoch**	**Total epochs**	**Total time**
RNN	RMSprop	1s	10	10s
CNN	Adam	1s	10	10s
Ts-RNN-Lin	RMSprop	13s	10	130s
Ts-CNN-Lin	Adam	6s	10	60s

7.5 Evaluation

All four models are evaluated using the HDFS system log dataset. The test settings are the same as the previous chapter with the addition of a new test in Section 7.5.3. The results of the RNN and CNN base models are listed in this section's tables to compare with the Ts models. Classification metrics including the accuracy, precision, recall, and F1 score are used for evaluation.

7.5.1 Evaluation Results

As mentioned earlier, a maximum duration $T_{\max}$ = 120s is a common choice to select eligible data for evaluation, as most samples fall within this duration as shown in Section 7.1. With $T_{\max}$ = 120s, a total of 101,245 samples are selected for training and evaluation. Among them, there are 6,422 faulty samples. A testing set containing 20% samples is used to evaluate all four models. Table 7.4 lists the performance of all four models. Note that the values in this table are not definitive as the models' final weights are subject to random initialization during the training process. Test results that are most reproducible are displayed in this table.

Table 7.4 Model performance on the test dataset.

	Accuracy	**Precision**	**Recall**	**F1-score**
RNN	99.96%	99.46%	99.92%	99.69%
CNN	99.96%	99.46%	99.84%	99.65%
Ts-RNN-ZOH	99.70%	97.81%	97.43%	97.62%
Ts-CNN-ZOH	99.87%	97.94%	100.00%	98.96%
Ts-RNN-Lin	99.71%	99.36%	96.03%	97.66%
Ts-CNN-Lin	99.96%	99.46%	99.84%	99.65%

From Table 7.4, the RNN, CNN, and Ts-CNN-Lin produce the best performance in terms of all four metrics and the difference is marginal. Ts-RNN-ZOH and Ts-RNN-Lin show the lowest F1 score, which is mainly attributed to a lower recall rate. It means the model gets more false negatives and misses some faulty samples. The contribution of timestamps seems to be insignificant to the detection tasks: Ts-CNN-Lin performs the same as CNN, while the two Ts-RNN models show degradation in performance. However, this is likely due to the models' performance being already at a very high level such that further improvements would be harder to achieve.

7.5.2 Results with Varying Duration Threshold

A list of larger $T_{\max}$ values ranging from 120s to 1920s is chosen for additional testing. Higher $T_{\max}$ gives longer samples that are shown to be difficult to classify in the previous chapter. A complete comparison is shown for the four models in Table 7.5 and Table 7.6, showing F1 score and accuracy respectively. Again, the most reproducible values among repeated tests are selected for presentation. In the F1 score table, Table 7.5 shows a distinctive difference among the tests. With more long samples (higher $T_{\max}$), the CNN and RNN base models both show decreased scores, while Ts-CNN-ZOH and Ts-CNN-Lin produce consistently strong performance with little degradation. At the maximum sample length of 1920s, the best performing two Ts-CNN models show a 3% advantage over the second-place base model. The two Ts-RNN models on the other hand show some degradation compared to the base RNN model, but have better performance with longer samples above 480s. The accuracy scores in Table 7.6 support a similar observation, but the difference among tests is less significant.

Table 7.5 Test F1 score with different T_{max}

T_{max}	F1 Score					
	RNN	**Ts-RNN-ZOH**	**Ts-RNN-Lin**	**CNN**	**Ts-CNN-ZOH**	**Ts-CNN-Lin**
120s	99.69%	97.62%	97.66%	99.65%	98.96%	**99.65%**
240s	97.80%	97.54%	95.14%	98.90%	**99.31%**	**99.31%**
480s	95.51%	95.42%	92.02%	98.59%	**99.02%**	**99.05%**
960s	93.64%	94.21%	95.54%	96.32%	**98.88%**	**98.84%**
1920s	91.69%	92.00%	92.79%	95.19%	**98.40%**	**98.40%**

Table 7.6 Test accuracy with different T_{max}

T_{max}	Test Accuracy					
	RNN	**Ts-RNN-ZOH**	**Ts-RNN-Lin**	**CNN**	**Ts-CNN-ZOH**	**Ts-CNN-Lin**
120s	99.96%	99.70%	99.71%	99.96%	99.87%	99.96%
240s	99.74%	99.70%	99.40%	99.84%	99.92%	99.92%
480s	99.52%	99.50%	99.10%	99.84%	99.89%	99.89%
960s	99.39%	99.44%	99.56%	99.61%	99.88%	99.88%
1920s	99.32%	99.34%	99.36%	99.56%	99.86%	99.86%

7.5.3 Repeated Tests

To further confirm these observations, each test setup in Table 7.5 is carried out 10 times. Every time the model is re-initialized with random weights. The accuracy results of each 10 repeated tests are presented as a box-and-whisker plot in Figure 7.5.1 and Figure 7.5.2, for the CNN and RNN variations, respectively. The box-and-whisker plot is a common graphical description of quantitative data. The box in the graph represents the interquartile range (IQR), covering the data points whose values are between the 75th and 25th percentiles, the middle 50%. The whiskers show the upper and lower 25th percentiles, except for the outliers that are indicated by individual dots. An outlier is determined if it lies on the outside of the 95% confidence interval of the distribution. The median performance is highlighted with a black cross mark and connected with dashed lines.

The improvement of adding timestamps is clearly revealed evident in the CNN case, as shown in Figure 7.5.1. Ts-CNN-Lin presents the highest median, best-case and worst-case F1 scores. Ts-CNN-ZOH underperforms for the 120s duration, but for the rest tests its performance is only marginally lower than the Ts-CNN-Lin. The base CNN model gets overall the lowest numbers of the three.

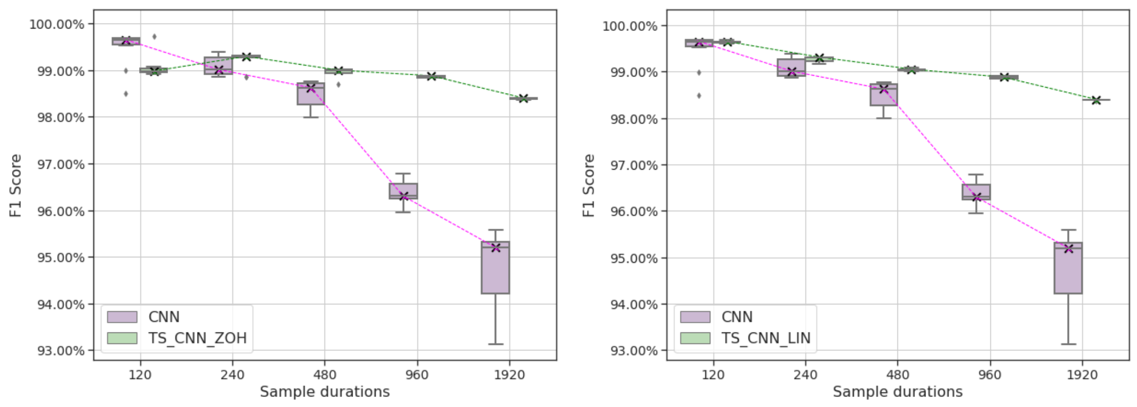


Figure 7.5.1 F1 score box plot comparison, CNN vs Ts-CNN-ZOH (left), RNN vs Ts-RNN-Lin (right)

For the RNN case, the addition of timestamps only shows its contribution for longer sequences, as shown in Figure 7.5.2. The repeated tests show that the base RNN has the lowest performance variation, while both Ts-RNN models have significantly larger values. The median performance of two Ts-RNN models is lower than the base RNN, other than for the longer sequence durations, i.e., the 960s and 1920s cases.

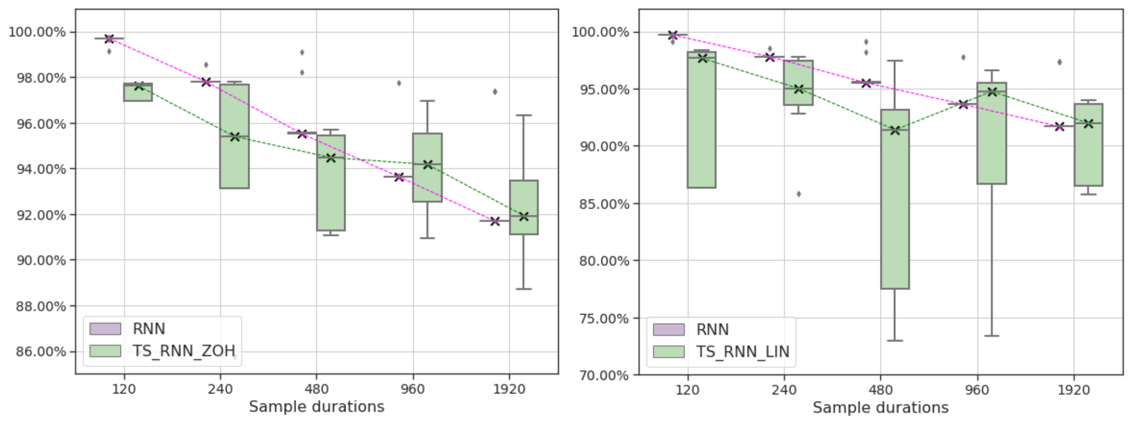


Figure 7.5.2 F1 score box plot comparison, RNN and Ts-RNN-ZOH (left), RNN and Ts-RNN-Lin (right)

7.5.4 Summary

The observation from tables and graphs in this section confirms that timestamps have a significant impact on distinguishing fault conditions. The temporal feature can be captured by the Ts-CNN variations, which improves upon the base CNN model. In particular, Ts-CNN-Lin demonstrates the best performance among all models tested. The RNN also benefits from the timestamps; however, it is only revealed through samples with longer durations.

7.6 Conclusion

This chapter investigated the effect of log timestamps on software fault diagnosis. The timestamps with irregular intervals are common in most types of log data, but few existing studies have examined or properly utilized them. Statistical and microscopic observations in this study reveal the major distributional differences when including timestamps, showing the potential of improving diagnosis performance.

The difficulty of including time information with log analysis is that the log tokens are categorical while the timestamps are numerical. The novel Ts models are proposed to solve the problem by resampling the embedding matrix within a deep learning framework. Different interpolation and extrapolation methods are investigated to perform the resampling process. The resample computation is implemented as a standard layer that is universally applicable to all types of neural networks that handle sequential inputs.

Two deep learning variations, Ts-CNN and Ts-RNN, are formulated under the Ts model structure. Each model can be configured to use ZOH or linear interpolations. The evaluation results demonstrate a significant improvement compared to the base models from the previous chapter, namely RNN and CNN. Ts-CNN-Lin model gives the best performance amongst the tested, showing a 3% improvement of F1 score followed by CNN. Timestamps show their contribution in the Ts-RNN models as well, although not as evident as the CNN case. It can be concluded that the timestamps contain distinguishing features that contribute to the detection of faults. Such features are successfully captured by the proposed Ts-CNN-Lin model.

One drawback of the Ts models, same as the sequential deep learning models in general, is that the data samples to be diagnosed must have a fixed duration. This is suitable for faults that exhibit complete faulty behaviour within a certain time period, but could be limited for latent conditions where the faulty behaviour takes a long time to present. The model’s input size can be set larger to include longer data samples and the decrease of performance has been significantly mitigated by using the Ts models, but the computational requirement is worth consideration. Large input size increases the load of resampling, which can add up the computational time for the Ts models. In practice, an appropriate time window that is long enough to capture the fault behaviour should be determined before configuring the diagnosis system.

Chapter 8 Conclusive Remarks

The problem of log data analysis is an emerging topic under the big data phenomenon. The system logs appear to be unstructured text data, but underneath is a mixture of categorical and numerical sequences. The difficulty of analysis can be mainly attributed to the categorical nature of log tokens, but also includes the complications of combing with other types of data, such as the timestamps. The system log typically comes in large quantities, and predictably their volume will continue to grow. Therefore, the analysis approach must consider the functionality as well as efficiency in order to process the data in large amounts.

Fault Detection and Diagnosis (FDD) is one important goal of data analysis. FDD systems can be roughly divided into two categories, namely model-based and signal-based. The former approach requires the mathematical model of the monitored system. In many cases obtaining such models are impractical, either because they are costly to build, or the system is too complex. For the systems that produce abundant data, data-driven models have been a cost-effective choice with the development of self-learning methods. These methods, represented by the recent deep learning models, are the key driving force to uncover the value of a large quantity of data. In turn, the availability of these technologies promotes the generation and harvesting of even more data as their benefits outweigh their development cost.

The software fault diagnosis for the Ford SYNC system is one good example in the big data environment. The SYNC's dedicated logging framework was developed to keep as much valuable information as possible and stores them as system logs. These log data play an important role in software development, especially customer acceptance testing and fault management. An automated fault diagnosis system using the system logs – as shown in this research – would be a great supplement to the current manual process and can improve the triage process. Other systems that produce log data could also benefit from such a diagnosis system, as demonstrated using the HDFS system log in this research.

Various approaches have been investigated in the research to tackle the complication of the log data. The problem originates from a software application, but the methodology is universal to any similar type of data. During the course of the research, many inspirations were drawn from other disciplines, such as bioinformatics, text analysis, language modelling, and signal processing. The journey was interesting and produced exciting achievements.

8.1 Investigated Methodologies

A few learning-based methodologies were investigated during the course of the research. They all fall into the broad category of machine learning. Three different approaches to decipher the fault features from training data were investigated, namely: sequence pattern mining (FDSPM), statistical machine learning (FDSML), and deep learning. They range from intuitive to abstract, representing the progress of this research from a simple to a more complex level. The methodologies are also loosely related to the evolution of existing studies, where earlier studies focus on hard evidence and explicit sequential patterns and more recent works obtain highly abstracted features through deep learning. In addition, the timestamps being an essential part of the logs are also investigated.

8.1.1 Fault Diagnosis via Sequence Pattern Mining (FDSPM)

The first proposed method is called Fault Diagnosis via Sequence Pattern Mining (FDSPM) and is presented in Chapter 4. The FDSPM is a novel approach combining data mining for pattern extraction and Bayesian inference for classification, the first of its kind in log data analysis literature. The concept is rather intuitive: it mimics test engineers' manual process of examining the log data by finding distinguishing log sequence patterns. Specifically, with log data samples that contain the same fault, the FDSPM looks for common patterns that occur across samples. The discovered patterns are then used to match the sequence in a new log sample. A pattern-based naïve Bayes classifier is then applied to determine the fault based on the matched patterns.

The learning of representative sequence patterns is achieved through sequence data mining. Specifically, a mining algorithm that searches common patterns within a set of samples is required. Such algorithms are commonly used in bioinformatics, such as DNA sequence research, and can be ported to log token sequences with little adaptation. Moreover, to fully utilize available data from the Ford SYNC system, a contrast mining approach is adopted. The contrast mining not only searches for common patterns from a set of faulty samples, but also excludes the patterns that appear in a contrast set. An algorithm called ConSGapMiner is selected for its efficiency and flexibility after reviewing various mining methods. It is implemented from scratch using Python codes and successfully produces the contrast patterns using the Ford log data.

The number of patterns discovered for each fault condition may vary, and they likely contain false positives due to the deficiency of training data. As a result, the weighting of each pattern becomes important as they may not contribute to a fault condition equally. A knowledge base is constructed to represent these weightings. Specifically, it is a matrix containing the conditional probabilities $P(p_i|h_i)$ for all patterns and all faults. The $P(p_i|h_i)$ values are obtained by observing the patterns' appearances in the training dataset. The extracted patterns and the knowledge base are the outcome of the learning phase and are to be used for diagnosis.

When diagnosing a sequence sample, a pattern-matching process is implemented to find the appearances of extracted patterns. The matching result – a list of true or false entries – is then utilized by a scoring mechanism to obtain the diagnosis result. Defining thresholding rules is applicable, but it turns out to be very complicated as there are multiple faults and a large number of patterns. Instead, the Bayesian classification is investigated to tackle the problem in a probabilistic approach. A naïve Bayes classifier is implemented to obtain the probability of a fault $P(h_i|p_1, p_2, \ldots)$ using the conditional probabilities $P(p_i|h_i)$ from the knowledge base. The output of the classifier is the fault candidate with the highest probability.

The FDSPM method evaluated on the Ford dataset demonstrates good performance in terms of detection and classification accuracy. The contrast mining strategy successfully addresses the lack of data issue – training each fault condition uses on three training samples on average and two in some cases. However, FDSPM's shortcomings are also clear. Because the tokens are processed as symbols, the computational requirement is very high even with an efficient sequence mining algorithm selected. This limits the FDSPM's ability to scale to a large amount of data samples and fault conditions.

8.1.2 Fault Diagnosis via Statistical Machine Learning (FDSML)

To address the efficiency issue, the Fault Diagnosis via Statistical Machine Learning (FDSML) approach is explored in Chapter 5. FDSML takes a completely different approach from the FDSPM method by using statistical features instead of explicit patterns to represent fault conditions. The statistical feature extraction turns the problem into a numerical domain and renders many machine learning classifiers applicable. This approach is not uncommon among log literature, but it was first applied to the Ford log data and revealed the lack of data issue.

Converting token sequences into numerical vectors is a process called feature vectorization. It is a manually defined process that depends on the type of the system and log data. Most vectorization techniques have a small computation footprint, often with a constant-time complexity $\mathcal{O}(1)$. As a result, they are particularly suitable for large amounts of data. This study has found that the term-frequency-inverse-document-frequency (TF-IDF) feature works ideally with the system log data.

The vectorization process converts the token sequences into fixed-length vectors, to which many well-established machine learning classification models can apply. Specifically, an MLP is implemented because of its flexibility to scale with the size of the input, i.e., the number of layers and the number of neurons within each layer are both configurable. A 3-layer MLP is found to produce the best accuracies on the available Ford log data.

FDSML’s computational efficiency improvement is very high, requiring less than one hundredth (1/100) of the FDSPM to train using the same Ford dataset. The saved time means quicker parameter tuning and better scalability. However, the accuracy result is slightly less than the FDSPM. The reason is mainly due to the lack of training data. An MLP contains thousands of trainable parameters to converge. The fact that each fault class only contains an average of three training samples is far from adequate. To evaluate a properly trained MLP model’s performance, artificially augmented log data samples are created by inserting known fault patterns into real log sequences. Training a similar MLP model on the enriched dataset produces near 90% accuracies, a significant improvement from the original test. This demonstrates that FDSML can be highly effective, but requires a rich dataset to fully train the model.

Despite FDSML’s satisfactory results, extracting statistical features from sequences loses apparent information: the samples are treated as a collection of tokens rather than a permutation. The sequential order is removed. To incorporate more information in the vectorization process, more sophisticated feature vectorization methods are needed, as well as machine learning models capable of processing such features.

8.1.3 Deep Learning

Deep learning is the third approach investigated for the log data analysis problem in Chapter 6 to handle larger input features and include the order information. The main components are the embedding process for advanced feature vectorization and the sequential neural network model for classification. Deep learning is an emerging topic in many engineering areas including log analysis. This study was based on two network types – the CNN and the RNN – and extended upon them to process log tokens and perform the fault detection task.

The embedding technique originates from language modelling research with the goal of capturing the semantics of words. Instead of extracting feature vectors from a whole sequence sample as used in the previous approach, embedding applies to each individual element of a sequence, such as a word in a paragraph. In this study, an adaptation of the embedding process is implemented for the log tokens. The tokens are represented by vectors in a high-dimensional space which, through visualization, reveals the statistics and relationships of the tokens.

A traditional machine learning model, such as the regular neural network (the MLP), is unable to directly process 2D data, so a specialized deep learning model is required. In particular, Convolutional Neural Network (CNN) and Recurrent Neural Network (RNN) models were investigated, implemented, and fine-tuned. The HDFS open log dataset was used for training and evaluation, since the Ford SYNC system was unable to provide enough data due to logistic issues. Both models have demonstrated good performance but also showed some degradation for longer sequence samples.

8.1.4 Timestamp Integration

Log data from both the Ford SYNC system and the HDFS log dataset contain timestamps with nonuniform intervals. This timestamp information appears to reflect some dynamics of the log data, but has been rarely investigated in related literature. Specifically, the histogram of the log sequences' lengths does not reveal any meaningful pattern, but the distribution of the log tasks' durations exhibits a clear bi-modal pattern. Based on this observation, a novel neural network layer that up-samples the log sequence is proposed to integrate log timestamp information. The up-sampling happens at the embeddings level and uses interpolation methods to even out the intervals between data points. The outcome of the up-sample layer is still in a 2D matrix form, so it is compatible with existing network structures, such as an RNN or a CNN, to carry out the classification task.

The new models integrating timestamps, called Ts-RNN and Ts-CNN, have demonstrated a significant performance improvement over the base RNN and CNN models in terms of accuracy, precision, and recall metrics. This result shows that the timestamps indeed contain distinguishing information of the faults and can be captured by the proposed Ts-models to improve detection performance.

The deep learning approach can be comprehensive as it is able to include a great amount of information from the log data, but there are limitations. The size of the model input must be pre-defined and cannot be arbitrarily long. The deep learning models are shown to have decreasing accuracy on longer samples, although this tendency can be improved by using the proposed time-stamped models. Time-stamped models may also encounter computational issues when processing long samples, due to the relatively heavy interpolation operation. In a nutshell, the deep learning approach brings advantages compared to the two previous methods, but the computation requirement needs to be well planned before applying the methodology.

8.2 Recommendations for Future Research

The possible future research directions that can be extended from this study are as follows.

During the course of the research, there was a shortage of data from the Ford SYNC system. Chapters 4 and 5 address the lack of data using carefully selected mining algorithms and data augmentation techniques. Cross-validation, another technique that helps relieve the lack of data issue, is also worth investigating as future work. Cross-validation performs training a certain number of times; each time the dataset is split into training and test set differently. As a result, cross-validation can accurately estimate a model’s performance with a limited amount of data.

The lack of data is partially due to the fact that the current fault management system is not designed to store a large amount of log data. The accessibility of the system from outside of the corporation is another impacting factor. Although the HDFS open dataset resembles the SYNC log in many ways and is used as a replacement, it could not match the complexity of the latter. Future work may consider streamlining the data collection of the SYNC system in order to meet the requirement of training large machine learning models.

Miscellaneous information in log messages, such as the variables in the log statement, were not considered in this study as they are not universal and their interpretation varies from system to system. They should be investigated as some distinctive features may be uncovered and may contribute to the diagnosis of system faults. However, it would require a more comprehensive dataset to verify and evaluate, again highlighting the bottle neck of data collection.

By the time of finishing this dissertation, the transformer, a new type of sequential neural network, started to emerge. The original work on transformer [109] claimed higher accuracy and less training time than other existing models on language translation tasks. Further work could include adapting the transformer to log data analysis, comparing it with CNN and RNN, and integrating it with the Ts models proposed in this research.

Implementing the diagnosis system on an embedded platform is another potential challenge. This is becoming a trend in many areas as intelligent devices are more computationally capable and commonly feature internet connectivity. Edge computing is an example that moves processing and computation of data from a central location, such as a server or the cloud, to a processor unit close to the system that is being monitored. Performing local diagnosis brings practical benefits when a large number of products are deployed and all generating data at the same time. Online learning of the diagnostic models can be carried out in a distributed form when a central model is difficult to generalize to all products.

The data-driven methodologies in this research are developed for analyzing system log data, which is a complex mixture of categorical sequence, numerical variables, and irregular timestamps. It would be interesting to apply the proposed methods in other domains using data of similar complexity and for applications that are not limited to software faults.

Appendix A ConSGapMiner Pseudo-code

The following pseudo-code is referenced from Chapter 6 in [81]. SMDS in the context is short form Semi-Minimal Distinguishing Subsequence, the intermediate variables to store sequence patterns.

A1. The wrapper function, `ConSGapMiner()`:

```
**Algorithm:** ConSGapMiner($pos$,$neg$,$g$,$\delta$,$\alpha$)
**Assumption:** $I$ is the alphabet list, $g$ is the maximum gap constraint,
    $\delta$ is the minimal support in $pos$, $\alpha$ is the maximal support in $neg$,
    a global set SMDS is used to contain the patterns generated by SMDS_Gen;
**Output:** $g$-MDS set $MDS$;
**Method:**
1:  $SMDS \leftarrow \{\}$;
2:  set $S$ to the empty sequence;
3:  SMDS_Gen($S$,$g$,$I$,$\delta$,$\alpha$);
4:  let $MDS$ be the result of minimizing $SMDS$ as described above;
5:  return $MDS$;
```

A2. The recursive function to grow the prefix tree, `SMDS_Gen()`:

```
Algorithm: SMDS_Gen($S$,$g$,$I$,$\delta$,$\alpha$);
Assumption: $S$ is a sequence, $g$ is maximum gap constraint, $I$ is the alphabet,
      $\delta$ is the minimal support for $pos$, $\alpha$ is the maximum support for $neg$;
      $CDS$ is a local variable storing the children distinguishing sequences of $S$;
      $SM$ is a global variable containing all computed distinguishing subsequences;
Method:
1:  initialize $CDS$ to {};
2:  for each $x \in I$ do
3:     let $S' = S.x$ (appending $x$ to $S$);
4:     if $S'$ is not a supersequence of any sequence in $CDS$ then
5:        $supp_{pos}$=Support_Count($S'$,$g$,$pos$);
6:        $supp_{neg}$=Support_Count($S'$,$g$,$neg$);
7:        if ($supp_{pos} \geqslant \delta$ AND $supp_{neg} \leqslant \alpha$) then
8:           $CDS = CDS \cup \{S'\}$;
9:        elsif ($supp_{pos} \geqslant \delta$) then
10:          SMDS_Gen($S'$,$g$,$I$,$\delta$,$\alpha$);
11: $SM = SM \cup CDS$;
```

A3. Support calculation and gap checking function `Support_Count()`:

```
Algorithm: Support_Count($S', g, D$);
Assumption: $g$ is the maximum gap, $S$ is the max-prefix of $S'$, the bitset array
      $BARRAY_S$ for $S$ is available, the bitset array $BARRAY_x$ for the final
      element $x$ of $S'$ is available, $D$ is the dataset;
Output: $supp_D(S', g)$ and $BARRAY_{S'}$ (the bitset array for $S'$);
Method:
1:  generate the mask bitsets $MaskS$ from $BARRAY_S$ for $g$ (stage 1 above);
2:  do bitwise AND of $MaskS$ and $BARRAY_x$ to get $BARRAY_{S'}$ (stage 2 above);
3:  let $count$ be the number of bitsets in $BARRAY_{S'}$ which contain 1;
4:  return $supp_D(S', g) = count/|D|$ and $BARRAY_{S'}$;
```

Appendix B Source Code for the Research

All source codes are available online as GitHub repositories as follows.

FDSPM in Chapter 4:

- https://github.com/hfyxin/log-data-analysis-data-mining

FDSML in Chapter 5:

- https://github.com/hfyxin/neural-network-with-dummy-data

Ts Models in Chapters 6 and 7:

- short version: https://github.com/hfyxin/Ts-models-log-data-analysis
- full version: https://github.com/hfyxin/time-sequence-alchemy-notebook

Bibliography

[1] "The value of Big Data: How analytics differentiates winners," *Bain*, Sep. 17, 2013. https://www.bain.com/insights/the-value-of-big-data/ (accessed Jan. 05, 2021).

[2] "Total data volume worldwide 2010-2024," *Statista*. https://www.statista.com/statistics/871513/worldwide-data-created/ (accessed Jan. 04, 2021).

[3] A. De Mauro, M. Greco, and M. Grimaldi, "What is big data? A consensual definition and a review of key research topics," in *AIP conference proceedings*, 2015, vol. 1644, no. 1, pp. 97–104.

[4] R. Gonzalez and R. Woods, *Digital Image Processing*, 4th edition. New York, NY: Pearson, 2017.

[5] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems 25*, Curran Associates, Inc., 2012, pp. 1097–1105.

[6] A. Yousefi, "Defect Localization using Dynamic Call Tree Mining and Matching and Request Replication: An Alternative to QoS-aware Service Selection," thesis, 2014.

[7] K. Naik and P. Tripathy, *Software Testing and Quality Assurance: Theory and Practice*. John Wiley & Sons, 2011.

[8] B. Parhami, "Defect, Fault, Error,..., or Failure?," *IEEE Transactions on Reliability*, vol. 46, no. 4, pp. 450–451, Dec. 1997, doi: 10.1109/TR.1997.693776.

[9] M. Cinque, D. Cotroneo, and A. Pecchia, "Event Logs for the Analysis of Software Failures: A Rule-Based Approach," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 806–821, Jun. 2013, doi: 10.1109/TSE.2012.67.

[10] T. M. Mitchell, *Machine Learning*. McGraw-Hill Education, 1997.

[11] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan, “Online system problem detection by mining patterns of console logs,” in *2009 Ninth IEEE International Conference on Data Mining*, 2009, pp. 588–597.

[12] J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu, and D. Zhang, “Learning to Log: Helping Developers Make Informed Logging Decisions,” in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Florence, Italy, May 2015, pp. 415–425. doi: 10.1109/ICSE.2015.60.

[13] Q. Fu *et al.*, “Where do developers log? an empirical study on logging practices in industry,” in *Companion Proceedings of the 36th International Conference on Software Engineering - ICSE Companion 2014*, Hyderabad, India, 2014, pp. 24–33. doi: 10.1145/2591062.2591175.

[14] A. Rabkin and R. Katz, “Chukwa: A system for reliable large-scale log collection,” in *Proceedings of LISA’10: 24th Large Installation System Administration Conference*, 2010, p. 163.

[15] A. Oliner, A. Ganapathi, and W. Xu, “Advances and Challenges in Log Analysis,” *Queue*, vol. 9, no. 12, p. 30:30-30:40, Dec. 2011, doi: 10.1145/2076796.2082137.

[16] Y. Liang, Y. Zhang, H. Xiong, and R. Sahoo, “An Adaptive Semantic Filter for Blue Gene/L Failure Log Analysis,” in *2007 IEEE International Parallel and Distributed Processing Symposium*, 2007, pp. 1–8.

[17] S. He, P. He, Z. Chen, T. Yang, Y. Su, and M. R. Lyu, “A Survey on Automated Log Analysis for Reliability Engineering,” *arXiv:2009.07237 [cs]*, Sep. 2020, Accessed: Feb. 24, 2021. [Online]. Available: http://arxiv.org/abs/2009.07237

[18] A. Bouloutas, G. Hart, and M. Schwartz, “On the Design of Observers for Fault Detection in Communication Networks,” in *Network Management and Control*, A. Kershenbaum, M. Malek, and M. Wall, Eds. Boston, MA: Springer US, 1990, pp. 319–338. doi: 10.1007/978-1-4613-1471-4_25.

[19] A. T. Bouloutas, “Modeling fault management in communication networks,” 1992.

[20] A. T. Bouloutas, S. Calo, and A. Finkel, “Alarm correlation and fault identification in communication networks,” *IEEE Transactions on Communications*, vol. 42, no. 234, pp. 523–533, Feb. 1994, doi: 10.1109/TCOMM.1994.577079.

[21] A. T. Bouloutas, G. W. Hart, and M. Schwartz, “Fault identification using a finite state machine model with unreliable partially observed data sequences,” *IEEE Transactions on Communications*, vol. 41, no. 7, pp. 1074–1083, Jul. 1993, doi: 10.1109/26.231938.

[22] A. Bouloutas, G. W. Hart, and M. Schwartz, “Simple finite-state fault detectors for communication networks,” *IEEE Transactions on Communications*, vol. 40, no. 3, pp. 477–479, Mar. 1992, doi: 10.1109/26.135715.

[23] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, “Pinpoint: problem determination in large, dynamic Internet services,” in *Proceedings International Conference on Dependable Systems and Networks*, Jun. 2002, pp. 595–604. doi: 10.1109/DSN.2002.1029005.

[24] M. Y. Chen, E. Kiciman, A. J. Accardi, A. Fox, and E. A. Brewer, “Using Runtime Paths for Macroanalysis,” in *HotOS*, 2003, pp. 79–84.

[25] Y.-Y. M. Chen, A. J. Accardi, E. Kiciman, D. A. Patterson, A. Fox, and E. A. Brewer, “Path-based failure and evolution management,” San Francisco, CA, 2004.

[26] M. Chen, A. X. Zheng, J. Lloyd, M. I. Jordan, and E. Brewer, “Failure Diagnosis Using Decision Trees,” in *International Conference on Autonomic Computing, 2004. Proceedings.*, New York, NY, USA, 2004, pp. 36–43. doi: 10.1109/ICAC.2004.1301345.

[27] A. Oliner and J. Stearley, “What Supercomputers Say: A Study of Five System Logs,” in *Dependable Systems and Networks, 2007. DSN’07. 37th Annual IEEE/IFIP International Conference on*, 2007, pp. 575–584.

[28] W. Xu, “System problem detection by mining console logs,” UC Berkeley, 2010.

[29] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan, “Mining Console Logs for Large-Scale System Problem Detection,” in *Proceedings of the Third conference on Tackling computer systems problems with machine learning techniques*, USA, Dec. 2008, p. 4.

[30] Y. Liang, Y. Zhang, A. Sivasubramaniam, R. K. Sahoo, J. Moreira, and M. Gupta, “Filtering Failure Logs for a BlueGene/L Prototype,” in *2005 International Conference on Dependable Systems and Networks (DSN’05)*, 2005, pp. 476–485.

[31] W. Jiang, C. Hu, S. Pasupathy, A. Kanevsky, Z. Li, and Y. Zhou, “Understanding Customer Problem Troubleshooting from Storage System Logs,” in *Proccedings of the 7th conference on File and storage technologies*, USA, Feb. 2009, pp. 43–56.

[32] J. Stearley, “Towards informatic analysis of syslogs,” in *2004 IEEE International Conference on Cluster Computing (IEEE Cat. No.04EX935)*, Sep. 2004, pp. 309–318. doi: 10.1109/CLUSTR.2004.1392628.

[33] S. E. Hansen and E. T. Atkins, “Automated System Monitoring and Notification with Swatch,” 1993.

[34] R. Vaarandi, “SEC-a lightweight event correlation tool,” in *IEEE Workshop on IP Operations and Management*, 2002, pp. 111–115.

[35] J. E. Prewett, “Analyzing Cluster Log Files Using Logsurfer,” presented at the Proceedings of the 4th Annual Conference on Linux Clusters, 2003.

[36] J.-G. Lou, Q. Fu, S. Yang, J. Li, and B. Wu, “Mining program workflow from interleaved traces,” in *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2010, pp. 613–622.

[37] Q. Fu, J.-G. Lou, Y. Wang, and J. Li, “Execution Anomaly Detection in Distributed Systems through Unstructured Log Analysis,” in *2009 ninth IEEE international conference on data mining*, 2009, pp. 149–158.

[38] D. Brand and P. Zafiropulo, “On communicating finite-state machines,” *Journal of the ACM (JACM)*, vol. 30, no. 2, pp. 323–342, 1983.

[39] G. Bochmann and C. Sunshine, “Formal methods in communication protocol design,” *IEEE transactions on Communications*, vol. 28, no. 4, pp. 624–631, 1980.

[40] A. Danthine, “Protocol representation with finite-state models,” *IEEE Transactions on Communications*, vol. 28, no. 4, pp. 632–643, 1980.

[41] A. W. Biermann and J. A. Feldman, “On the Synthesis of Finite-State Machines from Samples of Their Behavior,” *IEEE Transactions on Computers*, vol. C–21, no. 6, pp. 592–597, Jun. 1972, doi: 10.1109/TC.1972.5009015.

[42] X. Ning, G. Jiang, H. Chen, and K. Yoshihira, “HLAer : a System for Heterogeneous Log Analysis,” 2013.

[43] D. Lo, L. Mariani, and M. Pezzè, “Automatic Steering of Behavioral Model Inference,” in *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, New York, NY, USA, 2009, pp. 345–354. doi: 10.1145/1595696.1595761.

[44] X. Yu, P. Joshi, J. Xu, G. Jin, H. Zhang, and G. Jiang, “CloudSeer: Workflow Monitoring of Cloud Infrastructures via Interleaved Logs,” in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA, 2016, pp. 489–502. doi: 10.1145/2872362.2872407.

[45] G. Jiang, H. Chen, C. Ungureanu, and K. Yoshihira, “Multi-resolution Abnormal Trace Detection Using Varied-length N-grams and Automata,” in *Second*

International Conference on Autonomic Computing (ICAC'05), Jun. 2005, pp. 111–122. doi: 10.1109/ICAC.2005.42.

[46] I. Rouvellou and G. W. Hart, "Automatic alarm correlation for fault identification," in *Proceedings of INFOCOM'95*, Apr. 1995, vol. 2, pp. 553–561 vol.2. doi: 10.1109/INFCOM.1995.515921.

[47] K. Yamanishi and Y. Maruyama, "Dynamic syslog mining for network failure monitoring," in *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, 2005, pp. 499–508.

[48] M. Lim *et al.*, "Identifying Recurrent and Unknown Performance Issues," in *2014 IEEE International Conference on Data Mining*, Dec. 2014, pp. 320–329. doi: 10.1109/ICDM.2014.96.

[49] J.-G. Lou, Q. Fu, S. Yang, Y. Xu, and J. Li, "Mining Invariants from Console Logs for System Problem Detection," in *USENIX Annual Technical Conference*, 2010, pp. 1–14.

[50] I. Beschastnikh, Y. Brun, M. D. Ernst, A. Krishnamurthy, and T. E. Anderson, "Mining Temporal Invariants from Partially Ordered Logs," in *Managing Large-scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques*, New York, NY, USA, 2011, p. 3:1-3:10. doi: 10.1145/2038633.2038636.

[51] S. He, J. Zhu, P. He, and M. R. Lyu, "Experience Report: System Log Analysis for Anomaly Detection," in *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, Oct. 2016, pp. 207–218. doi: 10.1109/ISSRE.2016.21.

[52] X. Yu, S. Han, D. Zhang, and T. Xie, "Comprehending Performance from Real-world Execution Traces: A Device-driver Case," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA, 2014, pp. 193–206. doi: 10.1145/2541940.2541968.

[53] W. Xu, L. Huang, and M. I. Jordan, "Experience Mining Google's Production Console Logs," 2010.

[54] P. Bodik, M. Goldszmidt, A. Fox, D. B. Woodard, and H. Andersen, "Fingerprinting the Datacenter: Automated Classification of Performance Crises," in *Proceedings of the 5th European conference on Computer systems*, 2010, pp. 111–124.

[55] S. Lu, B. Rao, X. Wei, B. Tak, L. Wang, and L. Wang, “Log-based abnormal task detection and root cause analysis for spark,” in *2017 IEEE International Conference on Web Services (ICWS)*, 2017, pp. 389–396.

[56] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, “Detecting Large-scale System Problems by Mining Console Logs,” in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, New York, NY, USA, 2009, pp. 117–132. doi: 10.1145/1629575.1629587.

[57] Y. Liang, Y. Zhang, H. Xiong, and R. Sahoo, “Failure Prediction in IBM BlueGene/L Event Logs,” in *Seventh IEEE International Conference on Data Mining (ICDM 2007)*, Omaha, NE, USA, Oct. 2007, pp. 583–588. doi: 10.1109/ICDM.2007.46.

[58] T. Kimura, A. Watanabe, T. Toyono, and K. Ishibashi, “Proactive failure detection learning generation patterns of large-scale network logs,” *IEICE Transactions on Communications*, 2018.

[59] W. B. Cavnar and J. M. Trenkle, “N-gram-based text categorization,” in *Proceedings of SDAIR-94, 3rd annual symposium on document analysis and information retrieval*, 1994, vol. 161175.

[60] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*, 1 edition. New York: Cambridge University Press, 2008.

[61] H. Hamooni, B. Debnath, J. Xu, H. Zhang, G. Jiang, and A. Mueen, “LogMine: Fast Pattern Recognition for Log Analytics,” in *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, 2016, pp. 1573–1582.

[62] Q. Lin, H. Zhang, J. Lou, Y. Zhang, and X. Chen, “Log Clustering Based Problem Identification for Online Service Systems,” in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, May 2016, pp. 102–111.

[63] C. Yuan *et al.*, “Automated Known Problem Diagnosis with Event Traces,” in *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, New York, NY, USA, 2006, pp. 375–388. doi: 10.1145/1217935.1217972.

[64] Y. Liang, “Failure Analysis, Modeling, and Prediction for BlueGene/L,” PhD Thesis, Rutgers University-Graduate School-New Brunswick, 2007.

[65] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*, vol. 1. MIT press Cambridge, 2016.

[66] O. Russakovsky *et al.*, “ImageNet Large Scale Visual Recognition Challenge,” *arXiv:1409.0575 [cs]*, Sep. 2014, Accessed: Jul. 18, 2018. [Online]. Available: http://arxiv.org/abs/1409.0575

[67] M. Sundermeyer, R. Schlüter, and H. Ney, “LSTM Neural Networks for Language Modeling,” 2012.

[68] A. Graves, N. Jaitly, and A. Mohamed, “Hybrid Speech Recognition with Deep Bidirectional LSTM,” in *2013 IEEE workshop on automatic speech recognition and understanding*, 2013, pp. 273–278.

[69] Y. Zuo, Y. Wu, G. Min, C. Huang, and K. Pei, “An Intelligent Anomaly Detection Scheme for Micro-services Architectures with Temporal and Spatial Data Analysis,” *IEEE Transactions on Cognitive Communications and Networking*, vol. 6, no. 2, pp. 548–561, 2020.

[70] M. Du, F. Li, G. Zheng, and V. Srikumar, “DeepLog: Anomaly Detection and Diagnosis from System Logs Through Deep Learning,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, New York, NY, USA, 2017, pp. 1285–1298. doi: 10.1145/3133956.3134015.

[71] A. Das, F. Mueller, C. Siegel, and A. Vishnu, “Desh: Deep Learning for System Health Prediction of Lead Times to Failure in HPC,” in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, 2018, pp. 40–51.

[72] A. Das, A. Vishnu, C. Siegel, and F. Mueller, “Desh: Deep learning for hpc system health resilience,” *SC Poster Session*, 2017.

[73] N. Indurkhya, F. J. Damerau, and F. J. Damerau, *Handbook of Natural Language Processing*. Chapman and Hall/CRC, 2010. doi: 10.1201/9781420085938.

[74] B. Xia, Y. Bai, J. Yin, Y. Li, and J. Xu, “LogGAN: a Log-level Generative Adversarial Network for Anomaly Detection using Permutation Event Modeling,” *Inf Syst Front*, Jun. 2020, doi: 10.1007/s10796-020-10026-3.

[75] X. Zhang *et al.*, “Robust Log-Based Anomaly Detection on Unstable Log Data,” in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 807–817.

[76] S. Lu, X. Wei, Y. Li, and L. Wang, “Detecting Anomaly in Big Data System Logs Using Convolutional Neural Network,” in *2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing*

and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech), 2018, pp. 151–158.

[77] J. Han and M. Kamber, *Data Mining: Concepts and Techniques*. Elsevier, 2012.

[78] X. Ji and J. Bailey, “An Efficient Technique for Mining Approximately Frequent Substring Patterns,” in *Seventh IEEE International Conference on Data Mining Workshops (ICDMW 2007)*, Oct. 2007, pp. 325–330. doi: 10.1109/ICDMW.2007.121.

[79] R. She, F. Chen, K. Wang, M. Ester, J. L. Gardy, and F. S. Brinkman, “Frequent-subsequence-based prediction of outer membrane proteins,” in *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, 2003, pp. 436–445.

[80] X. Yun and Z. Yangyong, “BioPM:An Efficient Algorithm for Protein Motif Mining,” in *2007 1st International Conference on Bioinformatics and Biomedical Engineering*, Jul. 2007, pp. 394–397. doi: 10.1109/ICBBE.2007.104.

[81] G. Dong and J. Pei, *Sequence Data Mining*. Springer Science & Business Media, 2007.

[82] X. Ji, J. Bailey, and G. Dong, “Mining minimal distinguishing subsequence patterns with gap constraints,” *Knowledge and Information Systems*, vol. 11, no. 3, pp. 259–286, 2007.

[83] P. Domingos and M. Pazzani, “Beyond independence: Conditions for the optimality of the simple bayesian classifier,” in *Proc. 13th Intl. Conf. Machine Learning*, 1996, pp. 105–112.

[84] P. J. Liu *et al.*, “Generating Wikipedia by Summarizing Long Sequences,” *arXiv:1801.10198 [cs]*, Jan. 2018, Accessed: Apr. 17, 2021. [Online]. Available: http://arxiv.org/abs/1801.10198

[85] “Demonstration of k-means assumptions — scikit-learn 0.24.1 documentation.” https://scikit-learn.org/stable/auto_examples/cluster/plot_kmeans_assumptions.html#sphx-glr-auto-examples-cluster-plot-kmeans-assumptions-py (accessed Apr. 21, 2021).

[86] F. Pedregosa *et al.*, “Scikit-learn: Machine learning in Python,” *the Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011.

[87] M. Abadi *et al.*, “Tensorflow: A system for large-scale machine learning,” in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, 2016, pp. 265–283.

[88] A. Paszke *et al.*, “Automatic differentiation in pytorch,” 2017.

[89] P. He, J. Zhu, S. He, J. Li, and M. R. Lyu, “An Evaluation Study on Log Parsing and Its Use in Log Mining,” in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Jun. 2016, pp. 654–661. doi: 10.1109/DSN.2016.66.

[90] S. He, J. Zhu, P. He, and M. R. Lyu, “Loghub: A Large Collection of System Log Datasets towards Automated Log Analytics,” *arXiv:2008.06448 [cs]*, Aug. 2020, Accessed: May 20, 2021. [Online]. Available: http://arxiv.org/abs/2008.06448

[91] M. Du and F. Li, “Spell: Streaming Parsing of System Event Logs,” in *2016 IEEE 16th International Conference on Data Mining (ICDM)*, Dec. 2016, pp. 859–864. doi: 10.1109/ICDM.2016.0103.

[92] P. He, J. Zhu, Z. Zheng, and M. R. Lyu, “Drain: An online log parsing approach with fixed depth tree,” in *2017 IEEE International Conference on Web Services (ICWS)*, 2017, pp. 33–40.

[93] L. Tang, T. Li, and C.-S. Perng, “LogSig: Generating System Events from Raw Textual Logs,” in *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, New York, NY, USA, 2011, pp. 785–794. doi: 10.1145/2063576.2063690.

[94] Z. M. Jiang, A. E. Hassan, P. Flora, and G. Hamann, “Abstracting Execution Logs to Execution Events for Enterprise Applications (Short Paper),” in *2008 The Eighth International Conference on Quality Software*, 2008, pp. 181–186. doi: 10.1109/QSIC.2008.50.

[95] J. Zhu *et al.*, “Tools and Benchmarks for Automated Log Parsing,” Jan. 2019.

[96] X. Rong, “word2vec Parameter Learning Explained,” *arXiv:1411.2738 [cs]*, Jun. 2016, Accessed: Apr. 16, 2020. [Online]. Available: http://arxiv.org/abs/1411.2738

[97] C. Manning, “Lecture 1: Word Vectors I: Introduction, SVD and Word2Vec,” presented at the CS224n: Natural Language Processing with Deep Learning, Standford University, 2019. Accessed: Jun. 09, 2021. [Online]. Available: http://web.stanford.edu/class/cs224n/index.html

[98] T. Mikolov, K. Chen, G. Corrado, and J. Dean, “Efficient Estimation of Word Representations in Vector Space,” *arXiv:1301.3781 [cs]*, Sep. 2013, Accessed: Apr. 13, 2020. [Online]. Available: http://arxiv.org/abs/1301.3781

[99] J. Pennington, R. Socher, and C. Manning, “Glove: Global Vectors for Word Representation,” in *Proceedings of the 2014 Conference on Empirical Methods in*

Natural Language Processing (EMNLP), Doha, Qatar, Oct. 2014, pp. 1532–1543. doi: 10.3115/v1/D14-1162.

[100] S. S. Haykin, *Neural Networks and Learning Machines*, vol. 3. Pearson Upper Saddle River, 2009.

[101] S. Hochreiter, Y. Bengio, P. Frasconi, J. Schmidhuber, and C. Elvezia, "Gradient Flow in Recurrent Nets: the Difficulty of Learning Long-Term Dependencies," in *A Field Guide to Dynamical Recurrent Neural Networks*, IEEE Press, 2001.

[102] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber, "LSTM: A Search Space Odyssey," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 28, no. 10, pp. 2222–2232, Oct. 2017, doi: 10.1109/TNNLS.2016.2582924.

[103] Y. LeCun *et al.*, "Backpropagation Applied to Handwritten Zip Code Recognition," *Neural Computation*, vol. 1, no. 4, pp. 541–551, Dec. 1989, doi: 10.1162/neco.1989.1.4.541.

[104] R. Rehurek and P. Sojka, "Software framework for topic modelling with large corpora," 2010.

[105] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," *arXiv:1412.6980 [cs]*, Jan. 2017, Accessed: Apr. 24, 2020. [Online]. Available: http://arxiv.org/abs/1412.6980

[106] T. Tieleman and G. Hinton, "Lecture 6.5-rmsprop: Divide the Gradient by a Running Average of Its Recent Magnitude," presented at the COURSERA: Neural networks for machine learning, 2012.

[107] P. Prandoni and M. Vetterli, *Signal Processing for Communications*, 1st edition. Lausanne: EPFL Press, 2008.

[108] A. Quarteroni, R. Sacco, and F. Saleri, Numerical Mathematics, 2nd ed. Berlin Heidelberg: Springer-Verlag, 2007. doi: 10.1007/b98885.

[109] A. Vaswani et al., "Attention is All you Need," in Advances in Neural Information Processing Systems, 2017, vol. 30.