DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2016

Data Analysis of Minimally-Structured Heterogeneous Logs

An experimental study of log template extraction and anomaly detection based on Recurrent Neural Network and Naive Bayes.

CHANG LIU

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF COMPUTER SCIENCE AND COMMUNICATION

ROYAL INSTITUTE
OF TECHNOLOGY

Data Analysis of Minimally-Structured Heterogeneous Logs

An experimental study of log template extraction and anomaly detection based on Recurrent Neural Network and Naive Bayes.

CHANG LIU

Master's Thesis at CSC
Supervisor: Anders Holst
Examiner: Anders Lansner

Abstract

Nowadays, the ideas of continuous integration and continuous delivery are under heavy usage in order to achieve rapid software development speed and quick product delivery to the customers with good quality. During the process of modern software development, the testing stage has always been with great significance so that the delivered software is meeting all the requirements and with high quality, maintainability, sustainability, scalability, etc. The key assignment of software testing is to find bugs from every test and solve them.

The developers and test engineers at Ericsson, who are working on a large scale software architecture, are mainly relying on the logs generated during the testing, which contains important information regarding the system behavior and software status, to debug the software. However, the volume of the data is too big and the variety is too complex and unpredictable, therefore, it is very time consuming and with great efforts for them to manually locate and resolve the bugs from such vast amount of log data.

The objective of this thesis project is to explore a way to conduct log analysis efficiently and effectively by applying relevant machine learning algorithms in order to help people quickly detect the test failure and its possible causalities. In this project, a method of preprocessing and clustering original logs is designed and implemented in order to obtain useful data which can be fed to machine learning algorithms. The comparable log analysis, based on two machine learning algorithms - Recurrent Neural Network and Naive Bayes, is conducted for detecting the place of system failures and anomalies. Finally, relevant experimental results are provided and analyzed.

Contents

Chapter 1

Introduction

At Ericsson, the concept of Component Based Architecture (CBA) focuses on the implementation, integration and reuse of multiple appropriate off-the-shelf independent components into a well-defined software system [1] in order to meet certain design requirements for different applications. The primitive idea is the decomposition of the software design into individual functional or logical components, which are modularized and further interconnected via communication channels and application programming interfaces (APIs). Different components are designed and tested by separate development teams before being integrated together as a whole. The overall development flow is based on the idea of Continuous Integration (CI) [2], where all pieces are automatically built, compiled and merged without wasting additional time and efforts. Even though the build, development, testing and log collection of the CBA system are all easily automated, log analysis is still requiring huge human efforts and plenty of time, especially when engineers are trying to identify the implied causes of the system failures. With the scaling-up of the architecture and speeding-up of the software updates and testing, the logs containing significant information may often be ignored and too hard for test engineers to analyze.

Usually, when performing a log analysis, the test engineer consults log data from the test cases, the system under the test, as well as other tools and instruments such as traffic generators, vulnerability scanners, etc. They will consider the *time-stamp* of the failed test cases, and look into the different log files around this timestamp (normally half-hour before it, depending on specific cases). Counting on their prior experiences, the *system logs* (logs generated by system controllers instead of individual components) messages are examined for finding the unexpected behaviors which might reveal the cause of the failure. If the actual reason for the error is found out, a trouble report will be created and sent to component design team for debugging. Otherwise, the test engineers will inspect deeper into the components' logs, and communicate with or sit together with the design team attempting to fix the problem. However, in many situations, just for a single test suite (i.e., a set of test cases), such manual log analysis by human takes lots of time just aiming to get the possible cause of the failure, let alone after tens of times of speed-up on the test service's execution in the future. At that time, they will very likely be facing with more than one hundred test runs per day. Moreover, the log messages marked by *ERROR* may not offer enough information for debugging besides of its severity level. Even though there are some lists of known recurrent faults which can be referred to during analyzing, it is obviously not enough and very superficial since the recorded faults are limited but the variation and variety of the failure are considerable. Additionally, generally only the test cases which are failed are considered to be analyzed, and all the green ones (passed tests) are ignored even though they may contain some hidden errors which lead to the test failure.

By using machine learning methods plus a specifically designed raw log data analyzer, we could understand and analyze unstructured logs with multi-formity and huge volumes generated during the test service from various sources of the CBA system, in order to reduce human efforts on manually analysis on massive text files, detect faults' causes, and even provide deeper and clearer understanding of such massive log data. However, several major issues are stopping us from immediately implementing machine learning algorithms, understanding the log data and obtaining helpful information. First of all, every software component is independently designed, developed and tested by an individual team. After being composed as a entire software architecture, each of them will generate their own log files recording the execution of the same test service running on them. The difficulty lies in the proper comprehension of such a mixture model among different parallel processes for the same execution. Second, due to such separation of components' development, the log messages are heterogeneous because, under such independent unit development, different developers have different preferences when recording the executions using log messages. Third, the log messages are not fully structured, not well formatted and very complex so that simply parsing techniques will not work out extracting helpful information and further conducting complex analysis.

Consequently, the primary research questions are defined as following:

- How to represent the minimally-structured heterogeneous log data into decent formats in order to conduct learning process? More specifically, when faced with such log messages, how to identify the logs sharing the same template and further group them into a template group?

- What the statistical or machine learning models can be used in order to obtain the spatial/temporal patterns [3] of log messages? The spatial property is due to the different sources of the logs; the temporal feature is due to the chronological order of the logs.

- Based on the statistical superposition model [3] of the logs, how to evaluate the relationship between logs and detect anomalies for debugging?

The corresponding objective of this research is to answer these questions and provide usable prototyping for testing and log analyzing. Since the entire thesis project is actually based on a very open topic, there is no exact correct answers to this question. Consequently, the purpose of this project is to explore such area and conduct experimental assignments in order to find the suitable and feasible methods to address such practical problems in real life. The results of this project could provide different view to the practical log data generated from large scale software testing and valuable experience for the following work.

In this thesis project, a Log Extractor is designed to preprocess the collected raw log data, cluster them based on their similarity, extract the log template of each of the cluster, label the grouped clusters by unique identifications, and finally feed the labels as input to the next stage. In the second stage, Recurrent Neural Networks are constructed, trained and experimented in order to learn the sequential patterns from the label sequence obtained from previous stage. Meanwhile, a Naive Bayes model is built as a baseline for the comparison study. Finally, the experimental results from both models are analyzed. The general objective is to be able to identify the error causes and anomalies causing the test failures given new logs based on the learning obtained from previous logs.

The remaining part of this thesis is organized as the following: Chapter 2 describes the major problems in details in terms of the characteristics of the log data and how to tackle them. Additionally, several terminologies are defined. Chapter 3 offers the background study and literature review regarding the existing works about log analysis in general. In Chapter 4, the Machine Learning algorithms utilized to solve the problems are discussed in details, which are the Naive Bayes and Recurrent Neural Networks. The implementation and prototyping specifications are given in Chapter 5. And the experiments results and corresponding analysis are given in Chapter 6. Chapter 7 finalize this thesis report with conclusions, further discussions and future work.

Chapter 2

Background

First of all, several definitions of the terminologies regarding logs are offered in order to conveniently explain the ideas and the understand the log characteristics in the following of this report.

Definition 2.0.1. *Log Entry (or Entry)* is a single line of log message between two `Carriage Returns` . It may consist of a *Time-stamp*, a *Description* and other possible elements (like *Server-name* for system logs and *Severity-level* for some other specific logs from different components), such as Figures 2.6, 2.5, 2.2 and 2.3. It may only contain some free text without any *Time-stamp* such as Figure 2.4.

Definition 2.0.2. *Log Template (or Template)* is the common format of a group of log entries, which are sharing the same layout but filled with different parameters.

Definition 2.0.3. *Event* is defined as the basic element of the program executions, representing a single behavior or action of the program.

Remark 2.0.1. *Test Case (TC)* is an individual application running on the platform designed to test a certain aspect or requirement (like, traffic load or network connection) of the system.

Remark 2.0.2. *Test Suite (TS)* is a set of test cases integrated together for testing a group of relevant requirements which the system should meet.

For avoiding leaking Ericsson's data, it is better to only demonstrate a few log examples here as long as they are sufficient enough to illustrate my ideas. Same consideration is taken into account in *Chapter 6. Experiments and Results*.

2.1 Log Data Description

How to select the proper data sets for the data mining and analyzing process?

Figure 2.1 represents the structure of the set of log files belonging to a single test

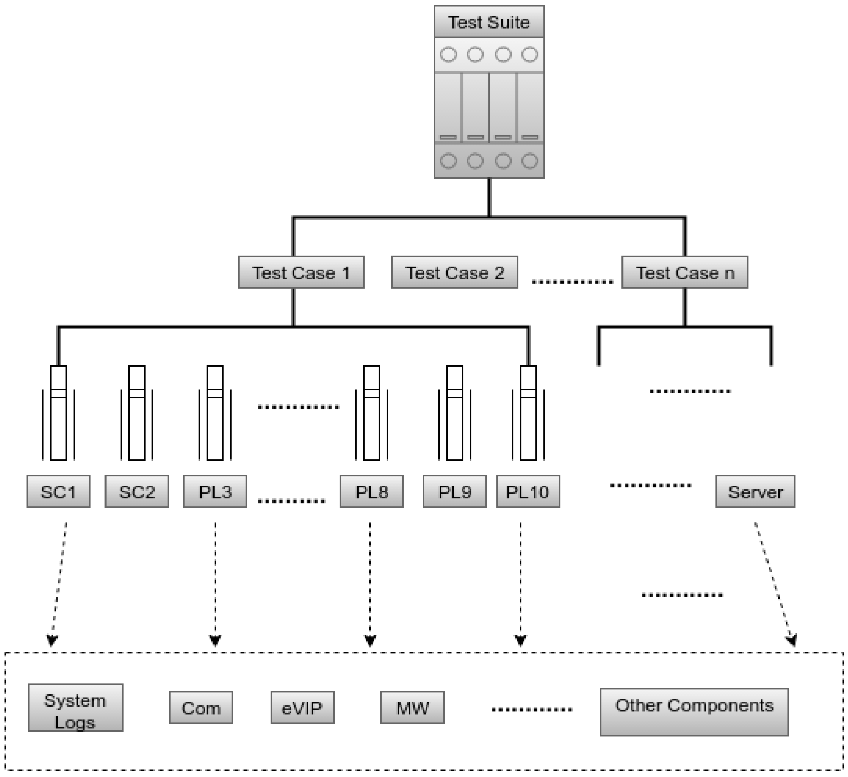


Figure 2.1: Log File Structure.

suite, which consists of a set of different test cases, each of which is executed for testing specific system aspects and requirements.

A test suite is released and run on a cluster of servers composing of several system controllers (SCs) and system payloads (PLs). Figure 2.1 shows an example of a system cluster with 10 servers, where there are two SCs (i.e., *SC1* and *SC2*) and eight PLs (i.e., *PL3 ~PL10*). The system logs, keeping tracks of the system behaviors during entire execution of the test suite, are generated at SCs. The logs for specific components are generated at different SCs and PLs (single server or a group of servers, depending on the system configurations) for the recording of various software components' behaviors. For conducting the data analysis, certain data set should be chosen for comparable and continuous analysis, in terms of different test cases and different components.

Since there are many kinds of components and different components are responsible for different functionalities, properly choosing a certain set of log files could make more sense in order to obtain the most significant information. Some of the components only create a single log file. Some of them, on the other hand, generate more than one type of log files. Since considering many components' logs would significantly increase the complexity of the work and go far beyond of the master thesis's scope, in this thesis project, only the system logs are considered. The detailed reasons are given in the following part of this chapter.

Additionally, there are several testing services running on the platforms, which are called "*Trains*" in Ericsson, such as *Green Train*, *Red Train*, *Blue Train* or *Nightly Train*. The "*Trains*" mentioned here represent the different test services in terms of different software development flows. For instance, the *Green Train* is the test service of the latest released version of the CBA components. The *Blue Train* is the latest CBA component versions waiting to be released. The *Red Train* is used to test and maintain the older versions. The *Nightly Train* is executed for daily testing in order to make sure a certain version is always ready and working well. For instance, *Nightly Blue Train* is running every night for testing the latest shipment of the software. Different services are executed periodically but in different scope and system configurations. The purpose is to get complete, sufficient and usable data for analysis. Nightly train might be the proper choice since it is executed more frequently so that the logs are easier to obtain and it is less frequently changed compared to other tests.

In future work, we could obviously scale it up by including more logs from different sources or having a larger data set. This data set plan is not fixed and will be varied based on the developing stage and analysis requirements.

2.1.1 Minimally-structured Logs

How to interpret the log messages with various types of formats into concrete representations (i.e., log templates)?

The system log entry is mainly minimally-structured as following:

Time-stamp	*Sever-name*	*Description*

Table 2.1: Message structure of *syslog*.

The *Time-stamp* indicates when this log message is generated, the *Sever-name* is the source server where this log is printed out, and, more importantly, the *Description* contains the most useful information for debugging but are with multiformities and hard to be interpreted. There are several different types of system log examples shown in Figure 2.6.

Most of the *Descriptions* related to system controllers are starting with a process name and its process id (PID) represented as `process-name[PID]` followed by a colon. However, most of the *Descriptions* generated from components written into the system logs are not with such format. They are starting with their component name or software module name, instead of process name, and without PID, like the examples shown in Figure 2.5. The first log entry in Figure 2.5 is from component *com*. And there are others from components or modules like *CoreMW*, *SMF* and *lde-brf-cmw*.

However, the log files of software components may be differed from the system log files in terms of the overall format, and they are even different from the component logs inside of system log files. For example, the messages presented in Figure 2.2 belongs to the component *CoreMW*'s log file. There is no *Sever-name* indicated in these log entries since, for the logs generated from different servers, they are stored within different folders under different server names, respectively. However, the thread numbers and severity level (such as `NOTIFY` in Figure 2.2) are shown in these log entries. Therefore, they are following the format as:

Time-stamp	*Number*	*Severity*	*Description*

Table 2.2: Message structure of *CoreMW*'s log.

Even further, the *Description* part of *CoreMW*'s logs are totally different from the *Descriptions* of system logs. For instance, instead of starting with a process name and PID like `process-name[PID]` in system logs, they start their *Descriptions* with `|MDTM` and `|BEGIN` as headers. The above structure of *CoreMW* is still different from the logs of component *vDicos* shown as the following:

Time-stamp	*Description*

Table 2.3: Message structure of *vDicos*'s log.

where only *Time-stamp* and *Description* are printed out. Such log message examples are shown in the Figure 2.3, where, for being properly displayed in this report, the message's *Time-stamp* is ignored and replaced by the wildcard symbol *. As shown in the graph, the difference is not only remaining in a single log entry but also reflected from a certain suite of logs.

Some log messages do not even have the corresponding *Time-stamp*. As shown in Figure 2.4, only the first line contains the *Time-stamp* whereas the following lines do not have any *Time-stamp* but rather belong to the save event associated with the first log entry. Even though the first line does have the *Time-stamp*, the time format is different from the *Time-stamp* within Figures 2.6, 2.5 and 2.2.

2.1.2 Heterogeneous Logs

No matter how the messages are structured, there is always a *Description* part existing in each of the log entries. However, the *Description* is neither uniformly well-formatted nor following a certain set of rules. The problem here is to understand this heterogeneous descriptive part and extract features and templates from it, in order to make the massive log messages well-regulated, since it contains most of the useful information needed for diagnosis. Even though the *Description* is indeed generated based on some formats rather than totally free text, their formats are totally different from each other depending on different programmers' preferences and purposes.

They are heterogeneous due to several reasons: **(i)** these messages are generated from different software components and different components are responsible for

```
Aug 14 17:12:47.956342 <1843003472> NOTIFY |MDTM: svc up event for SVCid
                                             =25, subscri. by SVCid =26 p
                                           we_id=1 adest=0002010f6d6c000d
Aug 14 17:12:47.966866 <1843003472> NOTIFY |MDTM: uninstall_tipc : svc i
                                             d=26,vdest id=65535
Aug 14 17:12:47.966873 <1835794445> NOTIFY |MDTM: svc down event for SVC
                                            id =26, subscri. by SVCid =25
                                             pwe_id=1 adest=0002010f6dda0
                                             050
Aug 14 17:12:47.976793 <1843134544> NOTIFY |BEGIN MDS LOGGING| PID=17341
                                             |ARCH=0|64bit=1
Aug 14 17:12:47.976951 <1843134544> NOTIFY |MDTM: install_tipc : svc id=
                                             26, vdest=65535
```

Figure 2.2: Log examples of component - *CoreMW*

```
* CdsvDirector::healthcheckCallback()
*
*    ----------------------- CDSvDirector status report ----------------
*     Running on node:  safAmfNode=SC-1,safAmfCluster=myAmfCluster
*     Active instance:  SC-2
*     Standby instance: SC-1
*
*     Controller status:
*     Instance is STANDBY.
*
*     Cluster is idle.
*    -------------------------------------------------------------------
```

Figure 2.3: Log examples of component - *vDicos.*

```
2016-02-15 10:16:45.505 IN initComponents
Start order for passive components:
 - MafMgmtSpiThreadContextService
 - MafOamSpiEventService
 - MafOamSpiCmRouterService
 - MafOamSpiTransactionService
 - VisibilityControllerComponent
```

Figure 2.4: Log examples of component - *com.*

different functionalities. Therefore, the log messages are more likely to present the information related to their corresponding components, with specific formats or

```
Aug  6 19:42:26 SC-1 com: COM_SA WARNING Received IMM relative distinguis
                     hed name to not existing IMM object <LicServer=.*>
...
Aug  6 19:42:47 SC-1 2015-08-06 19:42:47,492 ait:INFO ait_plugin_adapter:
                    184: install.sh >>> ERIC-IspC-CXP9022351_1-R3A02 imp
                     orted (type=Bundle)
Aug  6 19:42:47 SC-1 CMW: Invoked: [cmw-sdp-import /home/ait/repo/unpack/
                    plugin/CXP9022352-1_R3A02_depl/ERIC-ISP-C-I1-TEMPLAT
                    E-CXP9022352_1-R3A02/ERIC-ISP-C-I1-TEMPLATE-CXP90223
                     52_1-R3A02.sdp]
Aug  6 19:42:47 SC-1 SMF: smfBundleCheckCmd: [smfBundleCheckCmd --sdp=/tm
                     p/TEMP_SDP_lpN28w]
...
Aug  6 19:42:49 SC-1 lde-brf-cmw: admin operation, operationid: 0 (0)
Aug  6 19:42:49 SC-1 lde-brf-cmw: Handling operation 0
Aug  6 19:42:49 SC-1 lde-brf-cmw: reportActionResult: resultCode: 0
```

Figure 2.5: Examples of system logs

software module names; **(ii)** since different components are developed by different teams and different developers may have their own preferences when recording logs. For instance, some developer set label NOTIFY as *Notification* but some others may regard NO as *Notification*, whereas certain people see NO only as the English word "*No*"; **(iii)** like the log examples of component *vDicos*, some of the *Description* means nothing but a dotted line or a blank line, which are used for being properly displayed on the terminal; nevertheless, other *Descriptions* may contain important information. Furthermore, there is no doubt that it is barely possible to collect all the template of *Description* from the developers from a dozen of design teams.

The task is to group the same type of messages together and label them with unique ID representing same template of messages. Therefore, it can be the messages with same format expressing same meaning but filled with different parameters (like, IP address or some numerical values) grouped together. It can also be an individual type of message without any parameter. Or it can be a set of messages composing as a template group referring to a single event. Moreover, in the Figure 2.4, these logs will be merged together into a single line of log with the unique time-stamp.

```
Jan 31 01:33:01 SC-1 /usr/sbin/cron[17718]: (root) CMD (test -x /etc/cro
                     n.*/logrotate && /etc/cron.*/logrotate > /dev/null
                     2>&1)
Jan 31 01:33:26 SC-1 osafimmnd[9146]: NO Ccb 595465 COMMITTED (BRFC)
Jan 31 01:33:26 SC-1 osafimmnd[9146]: NO Ccb 595466 COMMITTED (BRFC)
Jan 31 01:33:26 SC-1 osafimmnd[9146]: NO Ccb 595467 COMMITTED (BRFC)
Jan 31 01:33:54 SC-1 syslog-ng[6844]: Log statistics; dropped='udp(AF_IN
                     ET(192.168.0.2:514))=0', dropped='tcp(AF_INET(192.1
                     68.0.2:5140))=0', processed='center(queued)=7071818
                     6', processed='center(received)=56572886', processe
                     d='destination(d_external)=699966', processed='dest
                     ination(d_mmas_oam_instance1_appserver)=1326', proc
                     essed='destination(d_mmas_oam_instance0_log)=134437
                     56', processed='destination(d_mmas_oam_instance1_lo
                     g)=13443756', processed='destination(d_mmas_oam_ins
                     tance0_appserver)=1337', processed='destination(d_m
                     mas_traffic_instance1_log)=13443564', processed='de
                     stination(d_mmas_traffic_instance0_log)=13443598',
                     processed='destination(d_mmas_traffic_instance1_app
                     server)=1381', processed='destination(d_auth)=20404
                     ', processed='destination(d_kernel)=241', processed
                     ='destination(d_mmas_external)=13445093', processed
                     ='destination(d_mmas_traffic_instance0_appserver)=1
                     373', processed='destination(d_messages)=2772391',
                     processed='source(s_mmas_external)=40334998', proce
                     ssed='source(s_local)=699966', processed='source(s_
                     external)=2092829', processed='source(s_mmas_oam_in
                     stance0_log)=13443756', processed='source(s_mmas_oa
                     m_instance0_appserver)=1337'
...
Jan 31 01:34:38 SC-1 TGC-4302[9496]: [I] Total Statistics:  start=08:34:
                     19.340   stop=01:00:00.000   sessions=0  duration=-
                     1442298880.00 sec  send-rate=-0.00 req/sec
Jan 31 01:34:38 SC-1 TGC-4302[9496]: [I]      label=TOTAL    id=N/A
                        protocol=N/A   send=0          send-failed=0
                           recv=0         fail=0      timeout=0      re
                     cv-unknown=0      active-sessions=0
Jan 31 01:34:38 SC-1 TGC-4302[9496]: [I] TGC Memory Usage: [ Allocated
                     135168B (132.00KiB) ] [ In Use  101296B (98.92KiB)
                     ] [ Free  33872B (33.08KiB) ]
```

Figure 2.6: Examples of system logs

2.2 Challenges and Limitations of Scope

As stated above in this chapter, several challenges and limitations of this thesis work are identified in the following:

- There are more than 20 software components running on the platform and each of them is logging its behavior independently. It is very unrealistic, in this master thesis project to consider them all or build a complex mixture model. Therefore, only the system logs are taken into account. It is the logs that contain most of the significant information about the system's behavior and it also gathers most of the valuable logs from the components.

- The logs are unstructured and heterogeneous as mentioned in the previous sections. It is mainly because they are generated from diverse software components, which serve various functionalities, are developed from different teams and are logging their behaviors in different ways.

- Due to the parallel running of multiple components on the platform, the logs generated from each of them could also be written into the final, single, serial system log in a randomly intervened manner. This brings a very big difficulty for machine learning methods to handle such complexity with limited amount of data.

- The amount of data available is also one of the major concerns and I have tried my best to obtain as much data as possible in order to train the machine learning model. This issue is further addressed in *Chapter 5: Design and Implementation.*

- The relevant machine learning algorithms, instead of rule based methods, are selected in this thesis project mainly for the reasons that: (i) there are already several existing works which are conducted based on rule-based methods for solving various log analysis problems; (ii) for the issues with such complexity, rule-based methods may not be capable of solving it and machine learning algorithms are worth of trying. There are so much to be explored in this area.

- What have been explored, discussed and analyzed in this thesis project is just a small part of Recurrent Neural Network (RNN) and the deep learning library Keras. There are much more in this area, which needs every relevant researchers and developers' knowledge and skills. RNN and Keras are very sophisticated and further exploration within this area could a very valuable future work to this thesis project.

Chapter 3

Literature Study on Log Analysis

In the this section, literature studies regarding the problems mentioned in previous section are conducted based on existing works. Their strengths and weaknesses under different usage cases are discussed. Possible solutions of our specific problems and their feasibility are proposed for further implementation.

3.1 Related Works

To some extent, the idea of extracting the log templates is to scan the alike log messages and inspect the similarities among them. The goal is to achieve the clusters, the greatest and reasonable extent, where the logs are similar to the ones within same cluster but dissimilar to the ones belonging to other clusters. Some people design the specific similarity functions in order to measure the similarity degree between two strings (i.e., two log messages). Some others apply the clustering algorithms by extracting and analyzing various features of the log messages in a data space and merge homogeneous logs into one template. Additionally, some people construct different hierarchical structures (e.g., tree data structure) and regard the templates as structuralized organizations, which means the logs are following certain structural formats. These techniques will be discussed in detail in the following.

One technique that could be implemented as a similarity function to detect the similarities between two strings is measuring their **Levenshtein distance** [4]. It represents the minimum number of single character edits (i.e., insertions, deletions and/or substitutions) required to change one string into another. In [5], Damerau improves the Levenshtein distance by including the transpositions of two adjacent symbols and produced a better distance detector, known as the **Damerau-Levenshtein distance**. In information theory, the **Hamming distance** [6] is the number of different characters between two strings with equal length. In other words, it represents the minimum number of substitutions required to change one string into another. However, by only implementing traditional distance measurement, it is hard to set the threshold for distinguishing whether two messages are similar or different on the basis of low-dimensional single characters, since messages are represented by a bunch of high-dimensional words. And there are still many other considerations (e.g., the term frequency, the type of a word indicating the tendency to belong to a template, the total length of a log message, the abstraction level of a log template, etc.) are ignored but having significant impact on the final results.

In [7], Kimura *et al.* classify the words (i.e., the term between two whitespaces) into five *classes* representing five levels of the tendency to belong to the log template, and give each of the class a weight value indicating its tendency. Then, a specific similarity function called ***LogSimilarity*** between a cluster C and a message X is defined as following:

$$\mathrm{LogSimilarity(C, X)} = \boldsymbol{w}^t\boldsymbol{x}/\boldsymbol{w}^t\boldsymbol{c}_x \tag{3.1}$$

where, $\boldsymbol{w} = [w_i]$ for $(i = 1, 2, ..., 5)$ is the weight vector of the tendency level to become a log template for each predefined class i. The $\boldsymbol{x} = [x_i]$ represents the number of words of class i in X and $\boldsymbol{c}_x = [c_{x,i}]$ represents the number of class-i's words appeared in both C and X. If the calculated highest log similarity is less than a predefined threshold E, a new template cluster C_{new} is created from X; otherwise, the message X is put into current cluster C. One shortcoming of Kimura *et al.*'s method is the ignorance of word's position information since the word's position does matter when considered to be part of a log template. A specific word normally has its unique position in one template but has different position in others.

In [8], Vaarandi presents an experimental clustering tool called **SLCT (Simple Logfile Clustering Tool)** for log data analysis. It can help the test engineers detect frequent formats from log files, to build log file profiles and to identify anomalies within log files. SLCT brings the notion of *dense data space* into discussion, which is assumed to contain the data points, each of which represents a log entry with categorical (non-numeric) attributes (i.e., the words of each log entry). One data point is n-dimensional if it contains n words, and the position information is taken into account. Instead of relying on the traditional distance based approaches, SLCT uses a density based method for clustering. It is because defining an appropriate distance function for categorical data is complicated, and the such concept becomes meaningless in a high-dimensional data space (please refer to [8] for details). SLCT, consequently, focuses on the dense regions in the data space and identifies the significant clusters, each of which is corresponding to a certain frequently occurring log template. SLCT is also able to detect the outliers, which are the data points dissimilar to these clustered log templates. However, the outlier defined here as infrequent log format do not necessarily be the anomalies or faults. Therefore, the outliers detected maybe irrelevant to the real failures. For demonstrating the concept of *dense region*, the log examples from [9] is given as Figure 3.1. For the sensitivity of 10%, the dense region of logs in Figure 3.1 is `(Server,1),(myserverl,2), (service,3), (down,5)` [9]. SLCT relies on the words' frequency too much but loses the sight of each word's class as indicated by Kimura *et al.* in [7].

```
Server myserver1 service XYZ down
Server myserver1 service ABC down
Server myserver1 service 123 down
```

Figure 3.1: Log examples for demonstrating *dense region.*

Several examples of SLCT's utilization can be found at [10], [11] and [12]. A visualization tool called LogView is developed in [13] in order to visualize the hierarchical structure of the clusters produced by SLCT applying treemaps. In [14], Reidemeister *et al.* applies Vaarandi's algorithm and text clustering to obtain features as clusters of similar patterns. However, even though the performance was significant, and the approach is robust and flexible with various attributes of the logs, preserving the features and matching them against log files encounter great overhead. Therefore, in [9], for clustering the extracted patterns, they use **a variant of Levenshtein distance** [4] as follows:

$$\boldsymbol{L}_n(x_1, x_2) = 1 - \frac{\boldsymbol{L}(x_1, x_2)}{\max(|x_1|, |x_2|)} \tag{3.2}$$

where x_1 and x_2 are two dense regions, which is originally proposed by Vaarandi in [8] as sets of frequent tokens with their absolute positions in the logs. Reidemeister *et al.* modifies it by using relative scales and heuristic parameter approximations. The notation $|x_1|$ represents the number of tokens in the region x_1. The function $L(x_1, x_2)$ calculates the distance on the basis of token-granularity between two regions x_1 and x_2, then is normalized by $max(|x_1|, |x_2|)$. In summary, the Equation 3.2 represents the similarity between two regions in terms of tokens.

In Vaarandi's work [15], the SLCT was compared with another data mining utility called **LogHound** [16], [17]. LogHound employs a frequent itemset mining algorithm (i.e., an Apriori-like breadth-first approach) by utilizing itemset *trie* data structure [18], [19] for discovering frequent patterns (templates) from event logs. The itemset is defined as the set of m word-position pairs of a certain log entry. For example, the log entry `Router * interface * down` can be represented as the following itemset: `{(Router,1), (interface,3), (down,5)}` [8]. The wildcards * are indicating the parameters within a template. According to [16], the *trie* is a memory-resident tree data structure which is guaranteed to contain all frequent itemsets. Each edge on the tree is labeled with the name of a certain frequent item, and each node contains a counter. Each path from root node to a leaf node represents a log template.

According to [12], both SLCT and LogHound are incorporated into Sisyphus log mining toolkit [20] developed at Sandia National Labs. Sisyphus is a collection of third-party software components providing three following distinct capabilities: "*automated generation of message types, automated grouping of time-correlated mes-*

sages, and interactive review of these results" [12]. However, currently, it is not available to download online anymore. Some shortages of SLCT and LogHound are still remaining. For instance, SLCT and LogHound rely on the words' frequency too much but have no considerations of words' type. Additionally, the outputted templates only appear more than N times, which is the tricky value to set and leaves the outliers disregarded.

Makanju *et al.* presents an another event log clustering tool called **IPLoM (Iterative Partitioning Log Mining)** [21], [22] by implementing three steps of hierarchical partitioning process, iteratively. Different from SLCT and LogHound, IPLoM does not rely on the Apriori algorithm, which is not efficient in computation for mining longer patterns [23] since it is mainly based on the frequent item set mining for finding association rules over transactional databases. However, IPLoM is able to find the clusters regardless of its instances appearing frequency. The three steps of partitioning are successively based on token count, token position, and searching for bijections, which are the bijective relationships between the set of unique tokens in two token positions selected (please refer to [21] for detailed algorithms).

In [12], Stearley proposes a novel use of the bioinformatic-inspired **Teiresias algorithm** [24], [25], [26] to automatically classify *syslog* messages and compares it with SLCT. It discovers all rigid patterns (called *motifs*, i.e., log templates in our definition) in categorical (non-numeric) data. More explicitly, Teiresias can find all the motifs M formed by characters within set C and don't-care wildcards * from a given set of strings X. There is at least a specificity of $L = W$ occurring at lowest K times in X, where K is a predefined value, L is the number of fixed characters within C, and W is the total width of the motif including wildcards. However, Teiresias algorithm is based on the character's level instead of the words, which may lead to the over-granularity issue due to the lack of sufficient abstraction. A usage of Teiresias algorithm is presented in [27] for intrusion detection system.

After experimental implementation and results analysis, Stearley concludes the comparison between Teiresias algorithm and SLCT in [12]. The Teiresias is shown to be more effective in automatically generating word-granular log templates and its careful sort order provides a near-optimal categorization of messages. Whereas requiring large memory space prevents it from scaling up to very large data set, analysis of logs under 10,000 lines would be more reasonable and acceptable. Even though SLCT's results is less-effective, it is not suffering from considerable memory usage. Taking a step back, Teiresias could potentially be modified to not include infrequent words in candidate motifs in order release the memory pains. However, compared with SLCT, it does not use word's position values, which are considered as very important information in forming log templates. In [21] and [22], Makanju *et al.* also conduct experiments and compare the outputs of IPLoM, SLCT, LogHound, and Teiresias on seven different event log files with over 1 million log events. Results show that IPLoM consistently outperforms the other algorithms.

In [28], a **SyslogDigest** system that can automatically transform and compress low-level minimally-structured *syslog* messages into meaningful and prioritized high-level network events. This system is designed with the combinational ideas of offline and online components. The *offline domain knowledge learning* component automatically extracts relevant domain knowledge from raw *syslog* data. Based on such acquired domain knowledge and other available information (e.g., temporal closeness of messages), the *online processing* component groups related messages into high-level events and present the prioritized results. In the first online step, the messages are decomposed into words separated by whitespace. For each type of messages, by following breath-first search tree traversal, a corresponding tree structure is constructed to express the template hierarchy. The top level message type is set as the root of the tree and all messages are associated with this message type. The most frequent combination of words is searched and associated with the parent as child, iteratively, until all messages have been checked. By following such tree structure, each branch from the root to one of the leaves represents one single log template.

In [29], the first version of **FDiag** is developed to extract the templates of log entries. The statistical correlation analysis is conducted after to establish probable causes and effect relationships for the interested faults. There are two types of tokens (i.e., words between whitespaces) are defined – *Constants* and *Variables*. A *Constant* is the sequence of tokens that comprise of English letters and punctuations while a *Variable* is the sequence of tokens comprising of English letters, punctuations and at least one digital number. The *Message Template Extractor* component [29] is used to extract all the *Constants* and *Variables* from the messages and create a standard data format based on them for further analyzing. The second generation of FDiag is introduced in [30], where a new workflow called *Space-R* is developed and added, which identifies the event pattern across a multi-node cluster system.

In the above literature study, several techniques for log template extraction from un- or minimally-structured log data are discussed. Each of them has its own strengths and weaknesses or usage limitations as well. The following points should be considered and balanced:

- Threshold: In many methods, a threshold value is needed to be decided but rather tricky. It can lead to different granularity and abstraction level of the final template results.
- Word frequency: The frequency of the words are always considered but it does not mean that the infrequent words are not parts of a log template.
- Word classification: The word classification is aiming to determine the level of tendency of a word to belong to the template. However, some methods

regard every word equally.

- Delimiter: Some works divide a log message into words based on whitespace whereas the separation level is too generic to extract meaningful templates. For example, in Figure 5.1, within the log message "`osafimmnd[9147]: NO Ccb 595467 COMMITTED (BRFC)`", if it is divided based on whitespace, `osafimmnd[9147]` is regarded as a word then the `PID 9147` will not be extracted; in contrast, if "`[`" and "`]`" are included as delimiter, such number can also be highlighted and treated individually without messed up with non-contextual symbols.

- Time-stamp: When considering the log template, we are aiming to decide the type of a log. However, logs with different time-stamp should not be considered as different logs. For example, the logs as "`Dec 18 * osafimmnd[*]: NO Ccb * COMMITTED (BRFC)`" and "`Dec 19 *osafimmnd[*]: NO Ccb * COMMITTED (BRFC)`" appearing in two different days are actually belong to a same template.

- Partition based on length: One observation is that "*the log messages that have the same line format are likely to have the same token length*" [21]. Initial partitioning of the log data set based on the token length is quite reasonable and will reduce the computational time and memory cost.

Among these existing works, many of the authors also continuously conduct further statistical analysis and pattern extraction based on the templates obtained. In the next chapter, two machine learning algorithms (i.e., Naive Bayes and Recurrent Neural Networks) are discussed.

Chapter 4

Machine Learning

In this chapter, the two major machine learning algorithms (Naive Bayes and Recurrent Neural Network) applied in this project are studied and discussed, mainly for the usage of sequence pattern learning and sequence prediction.

4.1 Naive Bayes

Naive Bayes is considered to be a simple technique for solving classification problems. It has been studied and widely used for several decades and has been used to deliver great achievements. By applying the Bayes' theorem, also known as the conditional probability, Naive Bayes is based an assumption that the value of a particular feature is independent of the value of any other feature, given the class variable. For solving different types of problems, the Naive Bayes actually includes some popular-used mechanisms are Gaussian Naive Bayes [31], Multinomial Naive Bayes [32], Bernoulli Naive Bayes [33], etc. Therefore, these techniques have been used widely in various applications such as image recognition, natural language processing, and information retrieval [34], [35], [36], [37].

Despite the naive design, Naive Bayes classifiers are working very well for solving many complex practical problems. The article [38], mainly focusing on illustrating the optimality of Naive Bayes, states that the *true reasons* for the surprisingly good performance of Naive Bayes classifiers are found. It is concluded that, even though there are strong dependences existing among attributes, Naive Bayes can still be optimal *if the dependences distribute evenly in classes, or if the dependences cancel each other out* [38]. Even though, according to [39], Naive Bayes are not as good as some other approaches, Naive Bayes classifiers can still be trained very efficiently based on some supervised learning. An advantage of Naive Bayes is that it could lead to the promising results by only requiring a small amount of training data, which makes it a good choice in our classification problem since the data samples available is considered to be insufficient.

The problem-specific Bayesian model will be given in the Chapter 5. In this section, we only discuss the theories and some applications.

4.1.1 Bayes' Theorem and Naive Bayes Model

In probabilistic theory and statistics, the Bayes' theorem, named after Reverend Thomas Bayes, describes the conditional probability of an event given another possible event that has happened. It was published by Pierre-Simon Laplace in a modern mathematical formulation [40], and further developed by Harold Jeffreys, who axiomatically combined Bayes' algorithm and Laplace's formulation [41]. Generally, a simple form of Bayes' theorem is represented as the following equation, mathematically:

$$P(Y|X) = \frac{P(Y)P(X|Y)}{P(X)} \tag{4.1}$$

where

- Y and X are two different events and $P(X) \neq 0$.
- $P(Y)$ and $P(X)$ are the probabilities of observing the event Y and X, respectively, without regard to each other.
- $P(Y|X)$, a conditional probability, is the probability of observing event Y given that X is true.
- $P(X|Y)$ is the (conditional) probability of observing event X given that Y is true.

An alternative form using Bayesian probability terminology can be represented as following:

$$\text{posterior} = \frac{\text{prior} \times \text{likelihood}}{\text{evidence}} \tag{4.2}$$

In many cases, for instance in Naive Bayes Model, the probability of event X, which is evidence in Equation 4.2, is a constant. Therefore, we wish to consider the impact of its having been observed on our belief in various possibilities of event Y. In such a situation the denominator of Equation 4.1, i.e., the probability of the given evidence X, is fixed. And the event Y is the actual variable. Bayes' theorem then shows that the posterior probabilities are proportional to the numerator as:

$$P(Y|X) \propto P(Y)P(X|Y) \tag{4.3}$$

After translating the above Equation 4.3 into English terms, the posterior is proportional to prior times likelihood [42], formulated as:

$$\text{posterior} \propto \text{prior} \times \text{likelihood} \tag{4.4}$$

For a more general Naive Bayes Model problem, imagine that instead of representing a single variable, X can be a vector representing n features (independent variables) as $X = (X_1, X_2, ..., X_n)$. Our task becomes

$$\begin{aligned} & P(Y = y|X_1 = x_1, X_2 = x_2, ...X_d = x_d) \\ = & \frac{P(Y = y)P(X_1 = x_1, X_2 = x_2, ...X_d = x_d|Y = y)}{P(X_1 = x_1, X_2 = x_2, ...X_d = x_d)} \end{aligned} \tag{4.5}$$

where $y \in \{y^1, y^2, ..., y^k\}$. And k is an integer specifying the number of classes in the problem

Considering $\prod_{i=1}^{n} P(X_i = x_i) = P(X_1 = x_1, X_2 = x_2, ...X_d = x_d)$ is fixed, we obtain the following equation:

$$\begin{aligned} & P(Y = y|X_1 = x_1, X_2 = x_2, ...X_d = x_d) \\ \propto\ & P(Y = y)P(X_1 = x_1, X_2 = x_2, ...X_d = x_d|Y = y) \\ \propto\ & P(Y = y)\prod_{i=1}^{n} P(X_i = x_i|Y = y) \end{aligned} \tag{4.6}$$

Given Naive Bayes Statistical Model in Equation 4.7, the parameters can be estimated from training examples in order to obtain the maximum posteriori probability. In other words, given a new test example $x = (x_1, x_2, ..., x_n)$, the output of the Naive Bayes classifier is

$$\begin{aligned} & arg \max_{j\in\{1...k\}} P(Y = y^j|X_1 = x_1, X_2 = x_2, ...X_d = x_d) \\ = & arg \max_{j\in\{1...k\}} \left(P(Y = y^j) \prod_{i=1}^{n} P(X_i = x_i|Y = y) \right) \end{aligned} \tag{4.7}$$

In our problem, it is with discrete probability distribution and discrete finite parameter space, which will be further illustrated in Section 5.2 of Chapter 5.

4.2 Artificial Neural Network (ANN)

Artificial Neural Networks (or ANNs) are generally regarded as a family of algorithms, modeled loosely after and inspired by the biological neural networks (in particular, the human brain) which are used to estimate or approximate (especially non-linear) functions. By having the advantage of absorbing a large number of unknown inputs, it is designed to recognize patterns. ANNs are generally represented as network of interconnected computational unit - "*neurons*". Neurons are capable of receiving input data either from external world or from previous neurons, process them and finally fire outputs to the next neurons or to the external world . The computations inside a neuron might be very simple (such as simple summing up of weighted inputs), or quite complex (each node might contain another complex model/network, such as LSTM, which will be discussed in Section 4.3.4). The connections between neurons have numeric weights that can be tuned based on experience (training data), making the neural networks adaptive to the inputs and capable of learning.

According to [43], the concept of *perceptron* was originally proposed by Warren McCulloch and Walter Pitts in their work published at 1943 [44] and further developed, in the 1950s and 1960s, by the scientist Frank Rosenblatt [45]. The most basic perceptron can be represented as following Figure 4.1.

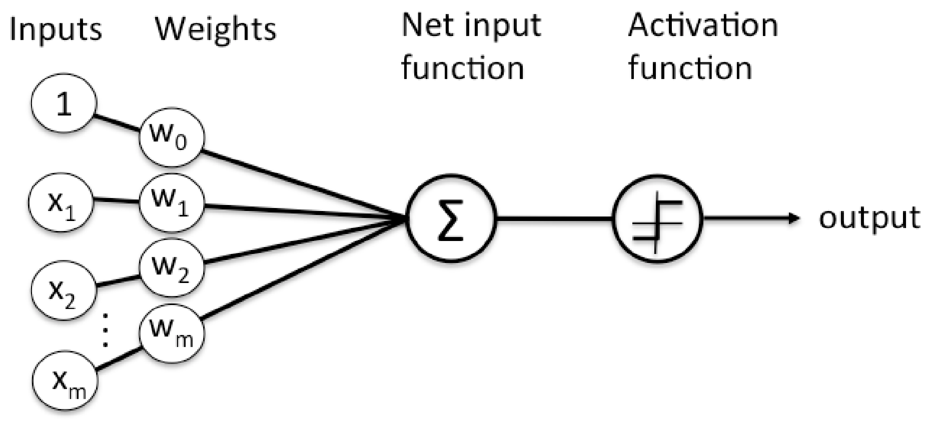


Figure 4.1: A single perceptron. Source: [46].

Mathematically speaking,

$$y = \phi\left(\sum_{i=1}^{m} w_i x_i + b\right) = \phi(\boldsymbol{w} \cdot \boldsymbol{x} + b) \tag{4.8}$$

where $\phi(\cdot)$ is the *activation function*, x_i for $i \in [1, m]$ are the inputs ($\boldsymbol{x}$ is the input vector), ω_i for $i \in [1, m]$ are the weights ($\boldsymbol{w}$ is the weight vector), and $b = w_0$ is the bias unit. In other words, each neuron performs a dot product with the inputs and the corresponding weights, adds the bias, applies the non-linearity (or activation function), and finally fires the output [47]. This can be regarded as the simplest decision making process.

4.2.1 Activation Function

The activation function describes the rate/frequency of the firing of the neuron. The simplest form of it is a binary one, i.e., either the neuron is firing or not, where Heaviside step function is the representative with linear transformation. However, it is the nonlinear activation function that allows neural networks to handle much more complex problems with accurate approximation of the desired function. There are several commonly used activation functions [47]:

- **Sigmoid** non-linearity is represented in the Equation 4.9 and is shown in Figure 4.2. It takes real numbers and maps them into the range of $(0, 1)$. With the input going to a very large negative number or large positive number, the sigmoid function will be saturated to 0 (tends not to fire) or 1 (tends to fire), respectively. This also leads to a near-zero gradient. In following Section 4.3.3, a problem called *vanishing gradient* will be discussed regarding how the multiplication of small gradients makes neural network hard to train. Additionally, sigmoid outputs are not zero-centered.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{4.9}$$

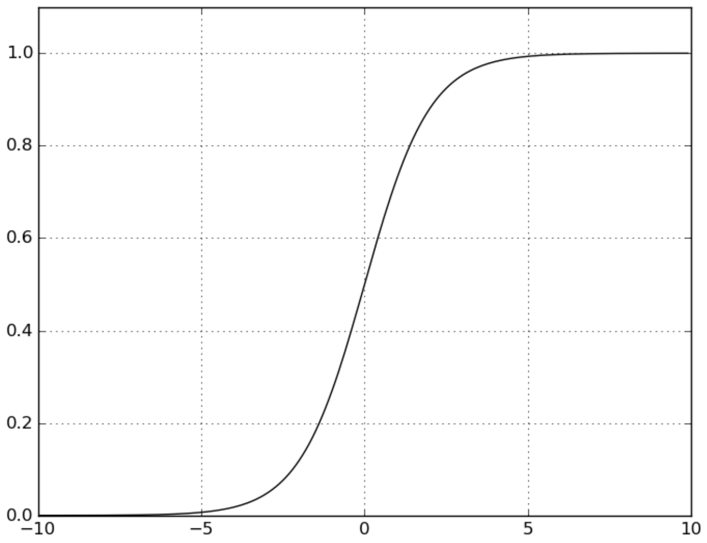


Figure 4.2: Sigmoid function.

- **Tanh** maps the real number input into the range of $(-1, 1)$, which is zero-centered as shown in Figure 4.3. Therefore, in practice, tanh non-linearity is always preferred to the sigmoid non-linearity [47]. In other parts, tanh, like the sigmoid, is also saturated and is just a scaled version of sigmoid.

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{2}{1 + e^{-2x}} - 1 = 2\sigma(2x) - 1 \tag{4.10}$$

- **Softmax**, in Equation 4.11, is normally taken as the activation function for the output layer in multi-class classification problem. It is linked to the cross-entropy loss. Given a K-dimensional vector $\boldsymbol{x}$ of arbitrary real values, softmax function will map every value of them to another real value in the range of $(0, 1)$, where all of them will be added up to 1.

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_{c=1}^{K} e^{x_c}} \quad \text{for} \quad i = 1, 2, ..., K \tag{4.11}$$

Some other activation functions that could be encountered in practices are ReLU (i.e., Rectified Linear Unit), Leaky ReLU [48], [49], Maxout [50], etc.

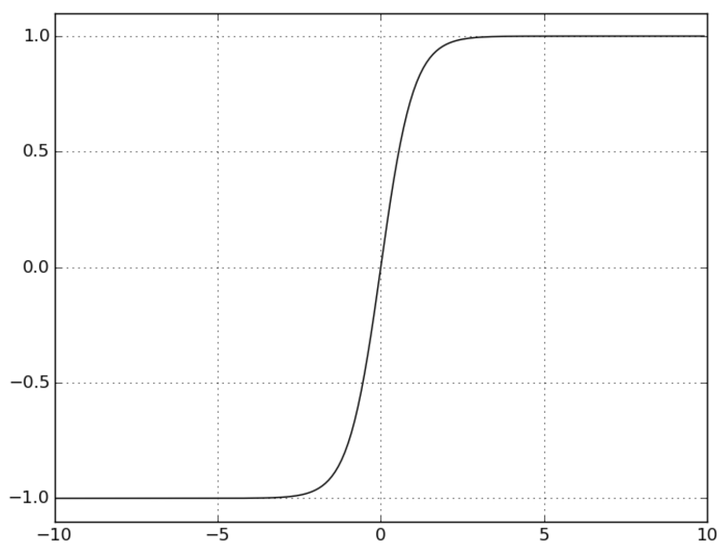


Figure 4.3: Tanh function.

4.2.2 Feed-forward Neural Networks

Feed-forward neural network, also often called Multi-Layer Perceptrons (MLP), is with the goal of approximating a function $f : X \to Y$, which is done by, for example, learning the value of the parameters $\boldsymbol{W}$ and $\boldsymbol{b}$ of the function $\boldsymbol{y} = f(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{W}, \boldsymbol{b})$. It is called *feed-forward* because the data flows from inputs $\boldsymbol{x}$, through the intermediate computations defined by f, and finally to the output $\boldsymbol{y}$ [51]. Instead of having one perceptron as in Figure 4.1, it is typically composed by several *layers* and there are no loops in the feed-forward network, which means information is always fed forward, never fed back. Some other networks contain loops, such as Recurrent Neural Network, which will be discussed in Section 4.3.

As one example of feed-forward neural network shown in Figure 4.4 [52], it is having three layers: (i) a n-unit input layer, (ii) a m-unit hidden layer, and (iii) a u-unit output layer. In previous example of a single perceptron, one perceptron is making its own decision based on its own input weighted by the parameters, and then generates a single output after further processed through activation non-linearity. In this example, perceptrons are lined-up together to form a layer, and different layers are stacked one by one forming a network. On each of the layer, every perceptron is making a decision depending on the inputs coming from the previous layer and its weights. In this way, layer by layer, perceptrons in the descendant layers are capable of making more complex decisions at more abstract level. Iteratively, the MLP can engage in very sophisticated problems and make much more hard decisions instead of just binary option.

Remark 4.2.1. The bias units are not shown in Figure 4.4.

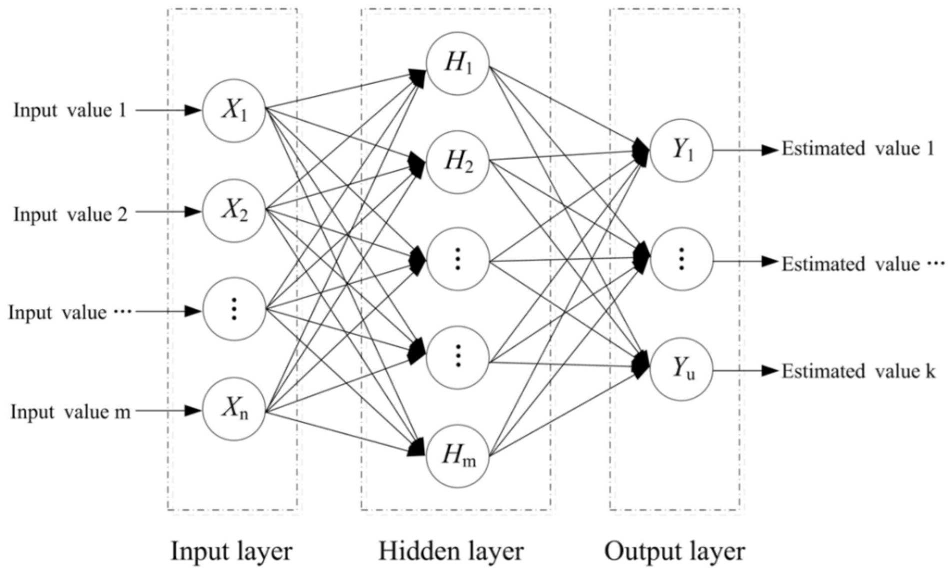


Figure 4.4: Feed-forward neural network (Multi-Layer Perceptrons). Source: [52].

For conveniently illustrating the feed-forward neural network mathematically, some remarks are given.

Remark 4.2.2. $L^{(i)}$ - the ith layer in the network. For example, in Figure 4.4, $L^{(1)}$ is the input layer, $L^{(2)}$ is the hidden layer, and $L^{(3)}$ is the output layer.

Remark 4.2.3. $s^{(i)}$ - the number of units (not counting bias unit) in layer $L^{(i)}$. For example, in Figure 4.4, $s^{(2)}$ is equal to m, which is the number of units in $L^{(2)}$.

Remark 4.2.4. $\boldsymbol{W}^{(j)}$ - the matrix of parameters controlling the function mapping from layer $j-1$ to layer j. If network has $s^{(j-1)}$ units in $L^{(j-1)}$ and $s^{(j)}$ units in $L^{(j)}$, the weight matrix $\boldsymbol{W}^{(j)}$ will be with dimensions of $[s^{(j)} \times s^{(j-1)}]$. $\boldsymbol{W}_i^{(j)} \in \mathbb{R}^{1\times s^{(j-1)}}$ is the vector of parameters controlling the function mapping from layer $j-1$ to the unit i of layer $L^{(j)}$. $W_{ir}^{(j)} \in \mathbb{R}$ is the parameter from the unit r in layer $j-1$ to the unit i in layer $L^{(j)}$. For example, in Figure 4.4, $\boldsymbol{W}^{(2)}$ is with space of $\mathbb{R}^{m\times n}$, $\boldsymbol{W}_1^{(2)}$ is with space of $\mathbb{R}^{1\times n}$, and $W_{11}^{(2)}$ is with space of $\mathbb{R}$.

Remark 4.2.5. $\boldsymbol{b}^{(j)} \in \mathbb{R}^{s^{(j)}\times 1}$ - the bias associated with the units in layer $L^{(j)}$, i.e., it will be associated with $\boldsymbol{W}^{(j)}$ during calculation. And $b_i^{(j)} \in \mathbb{R}$ is the bias contributing to the unit i in layer $L^{(j)}$. Bias units do not have inputs or connections going into them, since they always output the value $+1$.

Remark 4.2.6. $\phi(\cdot)$ - the activation function applied. Please refer to the Section 4.2.1 for more details.

Remark 4.2.7. $z_i^{(j)} \in \mathbb{R}$ - the total weighted sum of inputs to unit i in $L^{(j)}$. Then $\boldsymbol{z}^{(j)} \in \mathbb{R}^{s^{(j)}\times 1}$ is the weighted sum matrix in $L^{(j)}$ for all units.

Remark 4.2.8. $\boldsymbol{a}^{(j)} \in \mathbb{R}^{s^{(j)} \times 1}$ - the activation vector at $L^{(j)}$ for all units in the layer. Then $a_i^{(j)} \in \mathbb{R}$ is the activation of unit i in layer j. For example, $a_1^{(2)}$ is the activation of the first unit in the second layer. By activation, we mean the value which is computed and output by that node. As the output of L_j, $\boldsymbol{a}^{(j)}$ is also the input of its next layer - L_{j+1}.

Remark 4.2.9. $\boldsymbol{a}^{(1)} = \boldsymbol{x}$ - the activations of input layer is actually the input itself. Additionally, the activations of the last layer in the network are the outputs, i.e., $\boldsymbol{a}^{(l)} = \hat{\boldsymbol{y}}$ where l is the index of the last layer of a network with l layers.

By applying the notations given in Remarks 4.2.1 to 4.2.9, we could have the following representations of the neural network shown in Figure 4.4.

$$z_i^{(2)} = \sum_{k=1}^{n} W_{ik}^{(2)} x_k + b_i^{(2)} \qquad \text{for } i \in \{1, 2, ..., m\} \tag{4.12}$$

$$a_i^{(2)} = \phi\Big(z_i^{(2)}\Big) \qquad \text{for } i \in \{1, 2, ..., m\} \tag{4.13}$$

$$z_i^{(3)} = \sum_{p=1}^{m} W_{ip}^{(3)} a_p^{(2)} + b_i^{(3)} \qquad \text{for } i \in \{1, 2, ..., u\} \tag{4.14}$$

$$a_i^{(3)} = \phi\Big(z_i^{(3)}\Big) \qquad \text{for } i \in \{1, 2, ..., u\} \tag{4.15}$$

Remark 4.2.10. $\tilde{\boldsymbol{a}}^{(j)}$ - the extended input for layer L_{j+1}, which is the activation $\boldsymbol{a}^{(j)}$ at layer L_j plus the bias $+1$.

$$\tilde{\boldsymbol{a}}^{(j)} = \begin{bmatrix} \boldsymbol{a}^{(j)} \\ 1 \end{bmatrix} \quad \text{and} \quad \tilde{\boldsymbol{x}} = \begin{bmatrix} \boldsymbol{x} \\ 1 \end{bmatrix} \tag{4.16}$$

Remark 4.2.11. $\tilde{\boldsymbol{W}}^{(j)}$ - the extended weight matrix for layer L_j, which is the original matrix $\boldsymbol{W}^{(j)}$ plus the bias unit $b^{(j)}$.

$$\tilde{\boldsymbol{W}}^{(j)} = \begin{bmatrix} \boldsymbol{W}^{(j)} \\ b^{(j)} \end{bmatrix} \tag{4.17}$$

Then, by applying the Remarks 4.2.10 and 4.2.11, the more compact representations in matrix calculation of Equations 4.12 - 4.15 are:

$$\boldsymbol{z}^{(2)} = \boldsymbol{W}^{(2)} \cdot \boldsymbol{x} + \boldsymbol{b}^{(2)} = \tilde{\boldsymbol{W}}^{(2)} \cdot \tilde{\boldsymbol{x}} \tag{4.18}$$

$$\boldsymbol{a}^{(2)} = \phi\Big(\boldsymbol{z}^{(2)}\Big) \tag{4.19}$$

$$\boldsymbol{z}^{(3)} = \boldsymbol{W}^{(3)} \cdot \boldsymbol{a}^{(2)} + \boldsymbol{b}^{(3)} = \tilde{\boldsymbol{W}}^{(3)} \cdot \tilde{\boldsymbol{a}}^{(2)} \tag{4.20}$$

$$\boldsymbol{a}^{(3)} = \phi\Big(\boldsymbol{z}^{(3)}\Big) \tag{4.21}$$

which is called as *forward propagation.* In general, based on Remark 4.2.9, the feed-forward neural network are summarized as:

$$\boldsymbol{z}^{(j)} = \boldsymbol{W}^{(j)} \cdot \boldsymbol{a}^{(j-1)} + \boldsymbol{b}^{(j)} = \tilde{\boldsymbol{W}}^{(j)} \cdot \tilde{\boldsymbol{a}}^{(j-1)} \tag{4.22}$$

$$\boldsymbol{a}^{(j)} = \phi\Big(\boldsymbol{z}^{(j)}\Big) \tag{4.23}$$

for the network layer L_j.

By having the Equations 4.22 - 4.23, given an input vector $\boldsymbol{x} \in \mathbb{R}^{n\times 1}$, it is easy to obtain the output of the network: $\boldsymbol{y} \in \mathbb{R}^{u\times 1}$. However, the task of neural network is to approximate a function, which is a mapping from given $\boldsymbol{X}$ to $\boldsymbol{Y}$. How can we make sure that the outputted value is the actually true values wanted? In order to output the correct value, we should first understand how much difference there is between the *network estimation* and the *ground truth.* This is where the *loss function* is introduced.

4.2.3 Loss Function

Loss function (also referred to as the *cost function* or the *objective function*) represents the level of inaccuracy (difference) of our network prediction as to the ground truth. Intuitively, the loss will be low if the network is quite good at estimating the function mapping $\boldsymbol{x}$ to $\boldsymbol{y}$, otherwise, it will be high. Therefore, our task is to minimize the loss function in order to obtain a good estimation.

Given a set of training data set with N samples $\{(x^{(1)}, y^{(1)}), ..., (x^{(N)}, y^{(N)})\}$ and the corresponding network estimation output $\hat{y}^{(i)}$ for $i \in \{1, 2, ..., N\}$, based on the various of practical problem cases, there are different types of optional loss functions. The loss function used in this project is called *Categorical Cross-entropy (CCE)* as shown in Equation 4.24.

$$\text{CCE} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{K} y_c^{(i)} \log \hat{y}_c^{(i)} \tag{4.24}$$

where K is the number of output classes.

4.2.4 Back-propagation Algorithm

By having the loss function defined above, the desired function estimation can be done by adjusting weights in the network through back-propagation algorithm. The general idea of back-propagation algorithm can be represented, in abstraction, as the following three equations [46]:

$$\text{estimation} = \text{input} * \text{weight} \tag{4.25}$$

$$\text{error} = \text{ground truth} - \text{estimation} \tag{4.26}$$

$$\text{adjustment} = \text{error} * \text{weight's contribution to error} \tag{4.27}$$

Equation 4.25 refers to the Equations 4.22 - 4.23 in Section 4.2.2, which is used to obtain the network estimation using feed-forward computation. By utilizing the proper loss function mentioned in Section 4.2.3, the output error can be calculated and then used in Equation 4.26. Furthermore, *Gradient Descent* is introduced in order to obtain the errors based on each of the previous layers within the network based on the partial derivative of loss function in current layer, which is also the major procedure of back-propagation.

Gradient Descent, in neural network, is designed to find a local minimum of the loss function, which is parameterized by the network model's parameters $\boldsymbol{W}$ and $\boldsymbol{b}$, by applying the first-order optimization. The first partial derivatives of a loss function $J(W, b; x, y)$ with respect to the weight matrix $\boldsymbol{W}$ and bias matrix $\boldsymbol{b}$ are called the *gradient* of J. By updating the parameters in the opposite direction of the gradient, the loss function is gradually minimized and the network estimation will gradually converge to the ground truth value.

Notice: in the following analysis, the Remarks 4.2.2 - 4.2.11 are taken for granted. We also assume that the neural network has the layer $L^{(j)}$ for $j \in \{1, 2, ..., l\}$.

In this section, the loss function CCE, in Equation 4.24, is taken as an example for explaining the gradient descent, since this thesis project is actually a multi-class classification problem in specification. Considering the following loss function, in terms of one data point among N samples, which is one term of the total N terms summation of Equation 4.24:

$$J = -\sum_{i=1}^{K} y_i \log \hat{y}_i \tag{4.28}$$

where K is the number of output classes. Additionally, softmax function shown in Equation 4.11 is chosen as the output activation for multi-class classification problems, i.e.,

$$a_i^{(j=l)} = \hat{y}_i = \frac{e^{z_i^{(l)}}}{\sum_{c=1}^{K} e^{z_c^{(l)}}} \quad \text{for} \quad i = 1, 2, ..., K \tag{4.29}$$

where $a_i^{(l)}$ is the activation of the unit i in the last layer $L^{(l)}$, which is also the output of ith unit in the last layer. $z_i^{(l)}$ is the weighted sum at unit i in layer $L^{(l)}$, i.e., $a_i^{(l)} = \phi(z_i^{(l)})$. The gradient of the loss J over weight $W_{ir}^{(j)}$ is computed based on the chain rule, which is stated as following:

$$\frac{\partial J}{\partial \tilde{W}_{ir}^{(j)}} = \frac{\partial J}{\partial z_i^{(j)}} \frac{\partial z_i^{(j)}}{\partial \tilde{W}_{ir}^{(j)}} = \delta_i^{(j)} \tilde{a}_r^{(j-1)} \tag{4.30}$$

where

$$\begin{aligned}\delta_i^{(j)} = \frac{\partial J}{\partial z_i^{(j)}} &= \sum_{h=1}^{s^{(j+1)}} \frac{\partial J}{\partial z_h^{(j+1)}} \frac{\partial z_h^{(j+1)}}{\partial z_i^{(j)}} \\ &= \sum_{h=1}^{s^{(j+1)}} \frac{\partial J}{\partial z_h^{(j+1)}} \frac{\partial z_h^{(j+1)}}{\partial \tilde{a}_i^{(j)}} \frac{\partial \tilde{a}_i^{(j)}}{\partial z_i^{(j)}} \\ &= \phi'\left(z_i^{(j)}\right) \sum_{h=1}^{s^{(j+1)}} \tilde{W}_{hi}^{(j+1)} \delta_h^{(j+1)}\end{aligned} \tag{4.31}$$

is known as the *error term.* It is assumed that all units in layer L_{j+1} are connected to the unit i in layer L_j. For all hidden layers, the activation function is given as *tanh* as in Equation 4.10. Then, we have

$$\phi'\left(z_i^{(j)}\right) = \tanh'\left(z_i^{(j)}\right) = 1 - \tanh^2\left(z_i^{(j)}\right) = 1 - \left(a_i^{(j)}\right)^2 \tag{4.32}$$

Substituting the Equation 4.32 into Equation 4.31, we have

$$\delta_i^{(j)} = \left(1 - \left(a_i^{(j)}\right)^2\right) \sum_{h=1}^{s^{(j+1)}} \tilde{W}_{hi}^{(j+1)} \delta_h^{(j+1)} \tag{4.33}$$

and

$$\frac{\partial J}{\partial \tilde{W}_{ir}^{(j)}} = \tilde{a}_r^{(j-1)} \left(1 - \left(a_i^{(j)}\right)^2\right) \sum_{h=1}^{s^{(j+1)}} \tilde{W}_{hi}^{(j+1)} \delta_h^{(j+1)} \tag{4.34}$$

In particular, the gradient of the last output layer is computed in the following process, given the output activation function in Equation 4.29 [53]:

$$\frac{\partial J}{\partial \hat{y}_i} = -\frac{y_i}{\hat{y}_i} \tag{4.35}$$

$$\begin{aligned}\frac{\partial \hat{y}_i}{\partial z_q^{(l)}} &= \begin{cases} \dfrac{e^{z_i^{(l)}}}{\sum_{c=1}^{K} e^{z_c^{(l)}}} - \left(\dfrac{e^{z_i^{(l)}}}{\sum_{c=1}^{K} e^{z_c^{(l)}}}\right)^2 & i = q \\ -\dfrac{e^{z_i^{(l)}} e^{z_q^{(l)}}}{\left(\sum_{c=1}^{K} e^{z_c^{(l)}}\right)^2} & i \neq q \end{cases} \\ &= \begin{cases} \hat{y}_i(1 - \hat{y}_i) & i = q \\ -\hat{y}_i \hat{y}_q & i \neq q \end{cases}\end{aligned} \tag{4.36}$$

Combining Equations 4.35 and 4.36, we have the $\delta^{(l)}$:

$$
\begin{aligned}
\delta_i^{(l)} = \frac{\partial J}{\partial z_i^{(l)}} &= \sum_{c=1}^{K} \frac{\partial J}{\partial \hat{y}_c} \frac{\partial \hat{y}_c}{\partial z_i^{(l)}} \\
&= \frac{\partial J}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial z_i^{(l)}} + \sum_{c \neq i} \frac{\partial J}{\partial \hat{y}_c} \frac{\partial \hat{y}_c}{\partial z_i^{(l)}} \\
&= -y_i(1-\hat{y}_i) + \sum_{c \neq i} y_c \hat{y}_i \\
&= -y_i + \hat{y}_i \sum_{c=1}^{K} y_c \\
&= \hat{y}_i - y_i
\end{aligned}
\tag{4.37}
$$

The gradient for the weights in the layer L_l is:

$$
\frac{\partial J}{\partial \tilde{W}_{ir}^{(l)}} = \frac{\partial J}{\partial z_i^{(l)}} \frac{\partial z_i^{(l)}}{\partial \tilde{W}_{ir}^{(l)}} \tag{4.38}
$$

where $\tilde{W}^{(l)}$ is the weight matrix from the second last layer $L^{(l-1)}$ to the last layer $L^{(l)}$. Substituting the first-order derivative, with respective of $\tilde{W}$, of Equation 4.22 into Equation 4.38, we obtain:

$$
\frac{\partial J}{\partial \tilde{W}_{ir}^{(l)}} = (\hat{y}_i - y_i) a_r^{(l-1)} \tag{4.39}
$$

For simplicity, the previous back-propagation process can be represented in matrix formulations as:

$$
\frac{\partial J}{\partial \tilde{\boldsymbol{W}}^{(j)}} = \boldsymbol{\delta}^{(j)} \tilde{\boldsymbol{a}}^{(j-1)} \tag{4.40}
$$

where

$$
\begin{aligned}
\boldsymbol{\delta}^{(j)} = \frac{\partial J}{\partial \boldsymbol{z}^{(j)}} &= \frac{\partial J}{\partial \boldsymbol{z}^{(j+1)}} \frac{\partial \boldsymbol{z}^{(j+1)}}{\partial \boldsymbol{z}^{(j)}} \\
&= \frac{\partial \boldsymbol{z}^{(j+1)}}{\partial \boldsymbol{a}^{(j)}} \frac{\partial \boldsymbol{a}^{(j)}}{\partial \boldsymbol{z}^{(j)}} \boldsymbol{\delta}^{(j+1)} \\
&= \boldsymbol{\Lambda}^{(j)} \boldsymbol{W}^{(j+1)} \boldsymbol{\delta}^{(j+1)}
\end{aligned}
\tag{4.41}
$$

The activation derivative matrix $\boldsymbol{\Lambda}^{(j)}$ is defined as:

$$
\boldsymbol{\Lambda}^{(j)} = \frac{\partial \boldsymbol{a}^{(j)}}{\partial \boldsymbol{z}^{(j)}} = \begin{bmatrix} \frac{\partial a_1^{(j)}}{\partial z_1^{(j)}} & \frac{\partial a_2^{(j)}}{\partial z_1^{(j)}} & \cdots & \frac{\partial a_N^{(j)}}{\partial z_1^{(j)}} \\ \frac{\partial a_1^{(j)}}{\partial z_2^{(j)}} & \frac{\partial a_2^{(j)}}{\partial z_2^{(j)}} & \cdots & \frac{\partial a_N^{(j)}}{\partial z_2^{(j)}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial a_1^{(j)}}{\partial z_N^{(j)}} & \frac{\partial a_2^{(j)}}{\partial z_N^{(j)}} & \cdots & \frac{\partial a_N^{(j)}}{\partial z_N^{(j)}} \end{bmatrix} = \begin{bmatrix} \phi'\left(z_1^{(j)}\right) & 0 & \cdots & 0 \\ 0 & \phi'\left(z_2^{(j)}\right) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \phi'\left(z_N^{(j)}\right) \end{bmatrix}
\tag{4.42}
$$

where $N = s^{(j)}$ is the number of units in the layer L_j. For the output layer L_l, the $\boldsymbol{\delta}^{(l)}$ is given as:

$$\boldsymbol{\delta}^{(l)} = \frac{\partial J}{\partial \boldsymbol{z}^{(l)}} = \frac{\partial J}{\partial \hat{\boldsymbol{y}}} \frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{z}^{(l)}} = \hat{\boldsymbol{y}} - \boldsymbol{y} \tag{4.43}$$

To summarize the **back-propagation algorithm**, there are four major steps:

Step 1. Perform the **feed-forward** pass and compute the activations from the second layer $L^{(2)}$ to the last layer $L^{(l)}$ in order to obtain the predicted output.

Step 2. Compute the **output error** based on the predicted output and the ground truth value.

Step 3. **Back-propagate** the error from the last layer to the layer $L^{(2)}$.

Step 4. Compute the desired partial derivative, i.e., **gradients**, for each of the layer and update the weights.

$$\boldsymbol{W}^{(k)} = \boldsymbol{W}^{(k)} - \eta \frac{\partial J}{\partial \boldsymbol{W}^{(k)}} \tag{4.44}$$

where η is the *learning rate*. The higher the learning rate, the faster the learning process can be.

4.3 Recurrent Neural Network (RNN)

Traditionally, as what have been discussed in previous sections, the feed-forward neural networks do not contain any cycles. The information flows directly from input through hidden layers to the output, no backwards. Therefore, the number of computations performed through the feed-forward network is fixed, which only depends on the number of layers (and the number of units each layer) in the model. However, it has its own limitations when is is applied for solving many specific tasks. For example, if the task is to predict the next word in a sentence it is better to know which words came before it. However, in traditional network, it is assumed that all inputs are independent of each other. In this section, Recurrent Neural Network (RNN) is introduced for the idea of making use of sequential information.

RNNs are a particular branch of ANNs specially designed to recognize patterns in sequences of data or times series. It is because it has the capability of "memorization" which can be used to store the temporary information about what has been calculated. Ideally, arbitrarily long sequences could be remembered and utilized but, in practice, due to the restriction on the network complexity, computational resources and time, only a certain amount of steps (i.e., so called window size) can be looked back. Unlike within feed-forward networks, loops are formed within RNN, where internal states of the network can be created in order to store temporal memory.

By utilizing such memorizing advantage, the networks are capable of making more reasonable decisions based on previous observation and its own memory. For instance, in speech recognition, based on its memory during training and the given previous data or context, the network is able to predict the following content. More RNN applications can be in the post [54].

4.3.1 Recurrent Model

The Figure 4.5 shows how a simple RNN model, which contains a loop, is *unfolded* into a full (chain-like) network. The input x is weighted by the matrix U into hidden states s. Then, the hidden states s is iteratively calculated, through time, by weight W for certain time steps within the loop. The loop allows information to be passed from one step of the network to the next. Finally, the state s is transferred through V into output state y.

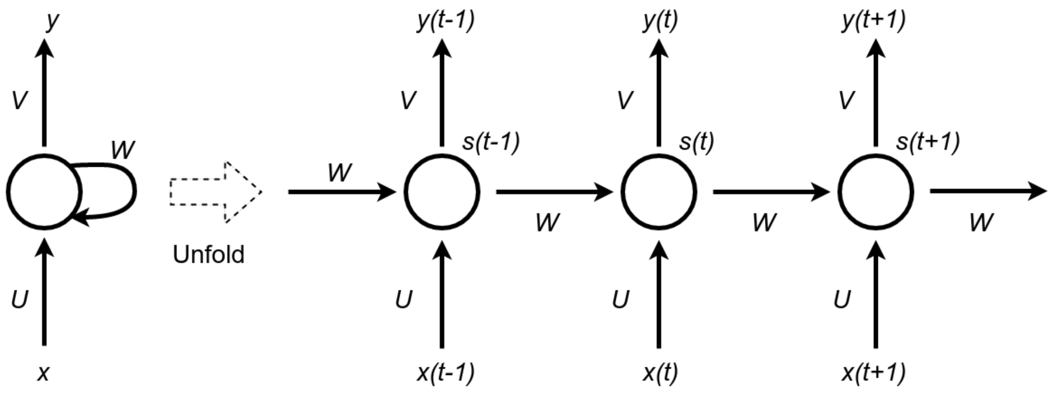


Figure 4.5: Simple example of unfolding RNN model.

- $x(t)$ is the input at time t.
- $s(t)$ is the hidden state at time t, which can be regarded as the place where "memory" is stored. $s(t)$ is calculated based on the previous hidden state and the current input, i.e.:
$$s(t) = \phi\Big(net_s(t)\Big) \tag{4.45}$$
where
$$net_s(t) = Ux(t) + Ws(t-1) + b_{in} \tag{4.46}$$
and ϕ it the hidden layer activation function, which is usually *tanh* or *sigmoid*, and the b_j is the bias on hidden states. The first hidden state $s(-1)$ is typically initialized to all zeroes.
- $y(t)$ is the output at time t and, for classification or symbolic data, generally, the output activation function is softmax function, i.e.:
$$y(t) = \psi\Big(net_y(t)\Big) \tag{4.47}$$

where, ψ is the output layer activation function, which is normally *Softmax*, and

$$net_y(t) = Vs(t) + b_{out} \tag{4.48}$$

where b_k is the bias unit on output units.

Simply speaking, a recurrent neural network can be thought of as multiple duplicates of the same network components folded together. Every fold is able to pass the data to the descendant states step by step based on time, like the chain-like network after unfolded from the loop, as shown in Figure 4.5. This is how RNN is capable of having temporal memory in Equation 4.45. Since it cannot have infinite loop and the unfolded length cannot be arbitrary as well, in practice, the network cannot remember the information from too many time steps ago.

For all the hidden states on the same layer but different time steps, same weight matrices U, V, W are shared. Therefore, on the same layer, the computations performed iteratively are the same between time steps. This greatly reduces the amount of parameters needed to be tuned [55]. Furthermore, one cell could be more complex than a single node. Consequently, Long Short Term Memory (LSTM), which is used for long-term time dependency, is discussed in Section 4.3.4.

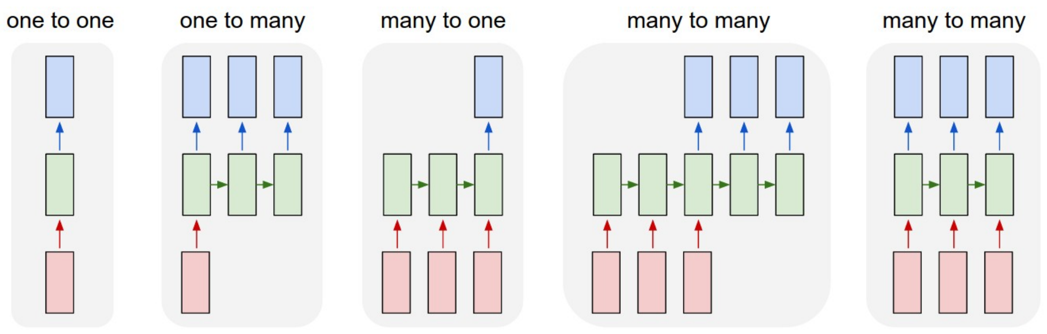


Figure 4.6: RNN models with various input and output. Source: [54].

The Figure 4.6 reveals how different practical problems can be mapped onto various structural RNN, where the pink cells represent input data (x in Figure 4.5), green cells represent hidden states (s in Figure 4.5), and blue cells represent output states (y in Figure 4.5). For example, when predicting the next word of a sentence, the input data would be a sequence of words and the output would be a single word to be predicted. The model selected could be the *many to one*. Another example is the translation model, where the input data is a sequence of words in source language (such as English) and the output data is a sequence of words (such as Swedish) in the target language. The model to be used could be *many to many*, where the input length and output length could be different as well.

4.3.2 Backpropagation Through Time (BPTT)

In Section 4.2.4, the back-propagation algorithm of error and gradient descent for traditional feed-forward neural network is discussed. However, the same approach cannot be directly applied on RNN model since the loop contained in the network based on time. In this section, the algorithm of *Backpropagation Through Time (BPTT)* [56] is introduced as an extension of normal back-propagation approach.

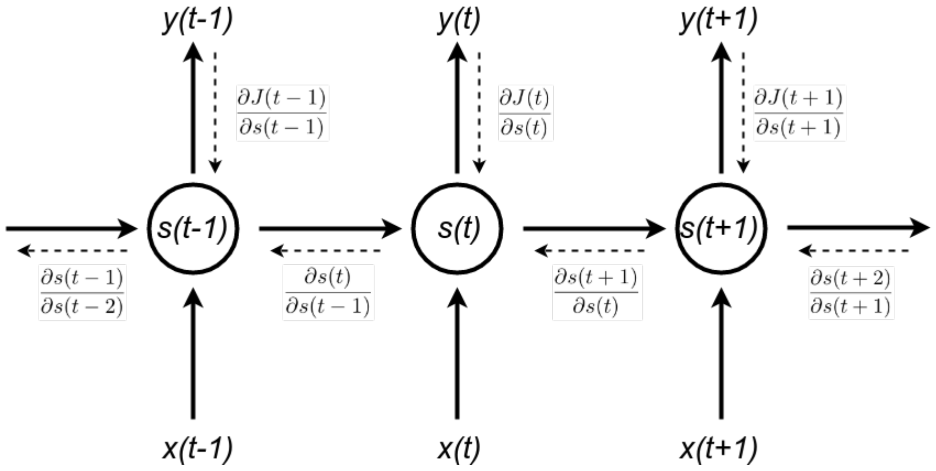

y(t-1)
y(t)
y(t+1)
$\frac{\partial J(t-1)}{\partial s(t-1)}$
$\frac{\partial J(t)}{\partial s(t)}$
$\frac{\partial J(t+1)}{\partial s(t+1)}$
s(t-1)
s(t)
s(t+1)
$\frac{\partial s(t-1)}{\partial s(t-2)}$
$\frac{\partial s(t)}{\partial s(t-1)}$
$\frac{\partial s(t+1)}{\partial s(t)}$
$\frac{\partial s(t+2)}{\partial s(t+1)}$
x(t-1)
x(t)
x(t+1)

Figure 4.7: Demonstration of Backpropagation Through Time.

Take the RNN model in Figure 4.5 as an example, the BPTT process is demonstrated in Figure 4.7. As stated in Equation 4.28, the original loss function is given as the categorical cross entropy expression over all classes. However, this equation is not taking time into account. Therefore, the particular cross entropy loss function based on time is given as:

$$J(t) = -\sum_{k=1}^{K} y_k(t) \log \hat{y}_k(t) \tag{4.49}$$

where $y_k(t)$ is the ground truth at time step t for class k, $\hat{y}_k(t)$ is the network prediction for class k, and K is the number of classes. The loss function $J(t)$ is also based on time step t. Then, the total loss is given as the sum of every losses at each of the time steps.

$$\begin{aligned} J &= \sum_{t=1}^{T} J(t) \\ &= -\sum_{t=1}^{T} \sum_{k=1}^{K} y_k(t) \log \hat{y}_k(t) \end{aligned} \tag{4.50}$$

where T is the number of time steps to be back-propagated. The gradient of loss $J(t)$ at time t over the weight V, which is the weight on the connection from hidden layer to output layer, is given as:

$$\begin{aligned}\frac{\partial J(t)}{\partial V} &= \frac{\partial J(t)}{\partial \hat{y}(t)}\frac{\partial \hat{y}(t)}{\partial net_y(t)}\frac{\partial net_y(t)}{\partial V} \\ &= \delta^{(y)}(t)s(t)\end{aligned} \tag{4.51}$$

where $\delta^{(y)}(t) = \hat{y}(t) - y(t)$ for the output layer. For the weight W, the gradient is calculated as:

$$\begin{aligned}\frac{\partial J(t)}{\partial W} &= \frac{\partial J(t)}{\partial \hat{y}(t)}\frac{\partial \hat{y}(t)}{\partial net_y(t)}\frac{\partial net_y(t)}{\partial s(t)}\frac{\partial s(t)}{\partial net_s(t)}\frac{\partial net_s(t)}{\partial W} \\ &= \sum_{\tau=0}^{t}\frac{\partial J(t)}{\partial \hat{y}(t)}\frac{\partial \hat{y}(t)}{\partial net_y(t)}\frac{\partial net_y(t)}{\partial s(t)}\frac{\partial s(t)}{\partial s(\tau)}\frac{\partial s(\tau)}{\partial net_s(\tau)}\frac{\partial net_s(\tau)}{\partial W} \\ &= \sum_{\tau=0}^{t}\delta^{(h)}(\tau)s(\tau-1)\end{aligned} \tag{4.52}$$

where, for $\frac{\partial s(t)}{\partial s(\tau)} = \prod_{j=\tau+1}^{t}\frac{\partial s(j)}{\partial s(j-1)}$ after applying chain rule,

$$\begin{aligned}\delta^{(h)}(\tau) &= \frac{\partial J(t)}{\partial \hat{y}(t)}\frac{\partial \hat{y}(t)}{\partial net_y(t)}\frac{\partial net_y(t)}{\partial s(t)}\left(\prod_{j=\tau+1}^{t}\frac{\partial s(j)}{\partial s(j-1)}\right)\frac{\partial s(\tau)}{\partial net_s(\tau)} \\ &= \delta^{(h)}(\tau+1)W\phi'\Big(net_s(t)\Big)\end{aligned} \tag{4.53}$$

Similarly, we can have the gradient of J over U as:

$$\frac{\partial J(t)}{\partial U} = \sum_{\tau=0}^{t}\delta^{(h)}(\tau)x(\tau) \tag{4.54}$$

where $\delta^{(h)}(\tau)$ is defined as same as Equation 4.53.

By having Equations 4.51, 4.54 and 4.54, the process of BPTT is essentially similar to the four steps described in Section 4.2.4 but with time taken into account.

4.3.3 Why RNN is Difficult to Train?

As stated in [57] by Pascanu *et al.* in 2012, there are two widely known issues causing the difficulty of training recurrent neural networks, i.e., the *vanishing gradient* and the *exploding gradient problems.*

In the work of Hochreiter's diploma thesis in 1991 [58], the reason of vanishing gradient problem is formally identified, which does not only affect deep feed-forward networks, but also the RNN. The major reason is that the RNN is trained and back-propagated based on time, where a very deep network is created with each of the layers representing one time step on the sequence [57]. In methods of gradient descent learning and BPTT as discussed in the Section 4.3.2, the back-propagation is created based on the chain rule of partial derivatives from the current time step back to the start time step. The gradient of each step is based on the activation function as shown in Section 4.2.1, which is normally ranging at $(-1, 1)$ or $[0, 1)$. Subsequently, the final output of back-propagation will be the multiplication of many small numbers of gradients from current layer to the front layers [57]. It means that the gradient will decrease exponentially with the number of layers during the process, and, consequently, the front layers will be trained very slowly.

Exploding gradient problem could be easier to solve compared to vanishing gradient problem. Instead of having very small values of gradients, the derivatives could also be a larger number, which leads to an exponentially large outcome after multiplication, i.e., exploding. The solution could simply be truncating or squashing these large values.

Literally speaking, traditional RNN, in practice, is struggling with remembering the long term dependencies. Take the sequence prediction as an example, given the following sentence "*I have been living in China for a long time. I can speak good ...*", the RNN could easily predict that "*Chinese*" might have a high probability as the next word. However, when there are plenty of sentences or even paragraphs and chapters existing between the two contents, it will be incredibly hard for RNN to make the reasonable prediction. More detailed calculation and analysis regarding *Why RNN is difficult to train?* or *Why long term dependency is hard with gradient descent?* can be found in [59], [60] and [57].

4.3.4 Long Short Term Memory (LSTM)

Long Short Term Memory (LSTM) is originally published by Sepp Hochreiter and Jürgen Schmidhuber in their work at 1997 [61]. It is an explicitly designed novel, efficient and gradient based method aiming to solve the problems of vanishing gradient, and also importantly, to remember the long-term dependency. The model shown in Figure 4.8 is a modified version of original LSTM, where the so-called *peephole connections* are added, meaning that all gate layers are taking the cell state into account. This is proposed by Felix A. Gers *et al.* in 2000 [62], [63].

From outside of the LSTM, it can be regarded as a black box, which is replacing a single tanh activation function in traditional RNN. However, from inside, it is very sophisticated and well-designed. In the Figure 4.8, the first gate after a sigmoid function is called *forget gate*, i.e., f_t at time t:

$$f_t = \sigma\Big(\boldsymbol{W}_f \cdot [C_{t-1}, h_{t-1}, x_t] + b_0\Big) \tag{4.55}$$

The input of the forget gate f_t is the piece-wise summation (concatenation) of the following data-flow: the output of the previous LSTM block h_{t-1}, the input for the current LSTM block x_t, the memory C_{t-1} of the previous block and finally a bias vector b_0. The f_t is ranging between 0 and 1 and will be multiplied with the previous memory C_{t-1}. The higher the value of f_t is, the more the previous memory will be kept. Therefore, the value of 1 means "all remembered" while 0 represents "all forgotten".

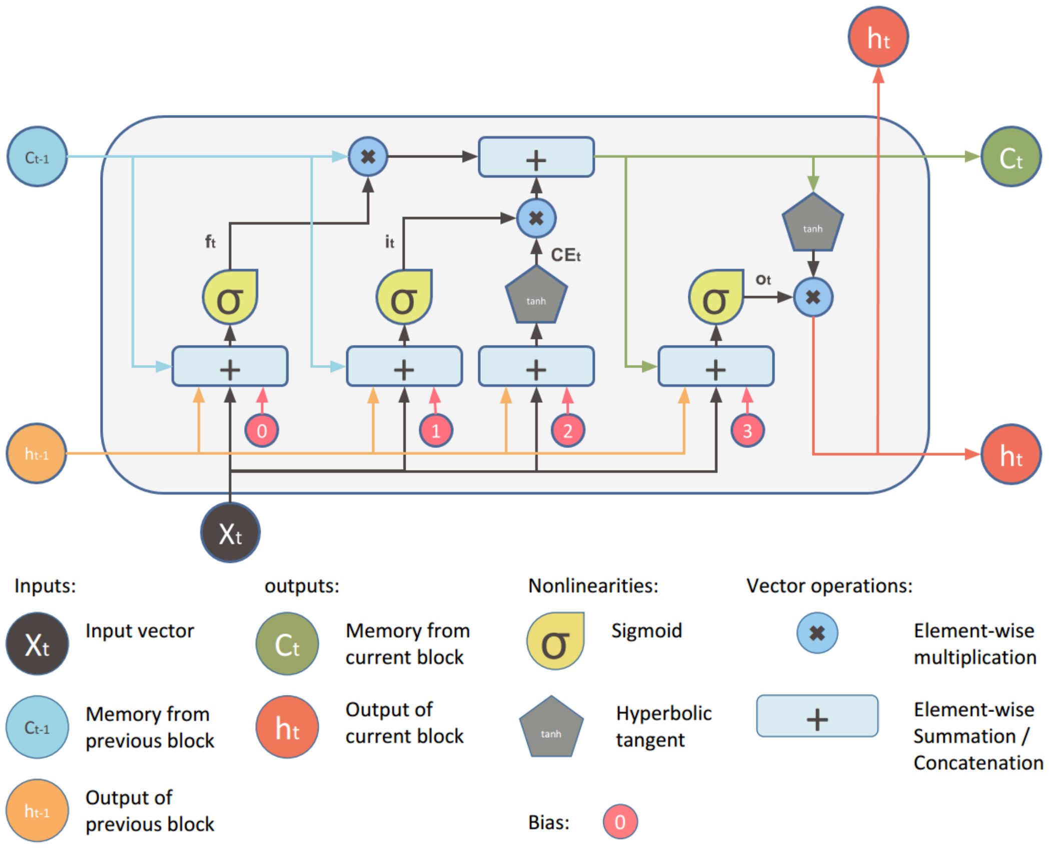


Figure 4.8: Model of Long Short Term Memory (LSTM) [61]. Source: [64].

The next component is representing the new memory to be updated, which is the multiplication between the results of *input gate* i_t in Equation 4.56 and a new candidate CE_t in Equation 4.57.

$$i_t = \sigma\Big(\boldsymbol{W}_i \cdot [C_{t-1}, h_{t-1}, x_t] + b_1\Big) \tag{4.56}$$

$$CE_t = \tanh\Big(\boldsymbol{W}_C \cdot [h_{t-1}, x_t] + b_2\Big) \tag{4.57}$$

The input gate i_t is a sigmoid layer, which has the same formulation as the forget gate, and CE_t is a tanh function layer.

Equation 4.58 shows how the new cell stated C_t is updated from the old cell state C_{t-1}.

$$C_t = f_t * C_{t-1} + i_t * CE_t \tag{4.58}$$

The output memory cell of this network is the sum of old memory through forget gate and the new memory through input gate.

Now, the *output gate* is represented as Equation 4.59.

$$o_t = \sigma\Big(\boldsymbol{W}_o \cdot [C_t, h_{t-1}, x_t] + b_3\Big) \tag{4.59}$$

The output unit is controlled by updated memory C_t instead of old memory. Additionally, it takes the current input x_t, previous output h_{t-1} and a bias unit b_3 into account.

$$h_t = o_t * \tanh(C_t) \tag{4.60}$$

Figure 4.8 shows a single LSTM block. In a RNN layer, every LSTM unit is also connected by utilizing the chain structure, where the previous cell state will be connected to the cell input of the next block and previous output will be connected to the next block as well. The way of how LSTM is controlling the information flow is just like how the pipes are piping the water. The gate layers in LSTM are just like the valves on the pipe, which controls how much water can pass through.

There are some more modified versions based on original LSTM having been developed in the past decades. A full comparison study can be found in [65].

4.3.5 Regularization and Dropout

Not enough training data or overtraining always lead to the problem of overfitting. The typical case of overfitting is: during the training process, the error on training data is gradually decreasing but the error on validation set turns to increase since the trained model is over fitting the training set but hardly working on the data beyond training set. There are several ways of avoiding overfitting such as early stopping, L1/L2 regularization (or, so called weight decay), and dropout.

L2 regularization is perhaps the most common form of regularization. It can be implemented by simply adding the sum term $\frac{\lambda}{2n}\sum_w w^2$ of every weight w to the objective function, where λ is the regularization strength and n is the number of training data. It is common to see the factor of $\frac{1}{2}$ because the gradient of this square term with respect to the parameter w is simply λw instead of $2\lambda w$, where 2 and $\frac{1}{2}$ are compensated by each other. During parameter update step of gradient descent, using the L2 regularization introduces another term $-\lambda * w$ to the equation. Its effect is reducing the weight linearly, which is why such regularization is also called weight decay. What is avoided with weight decay is the dramatically different size orders of weights. Now, introducing the regularization which prevents extremely high weights among all weights, the network is thus forced to learn the general patterns rather than the exceptions.

L1 regularization is another common form of regularization, where the term $\frac{\lambda}{n}\sum_w |w|$ is added to the cost function. However, in practice, L2 regularization is outperforming L1 to a great extent. It is also possible to combine the L1 regularization with the L2 regularization: $\frac{\lambda_1}{n}\sum_w |w| + \frac{\lambda_2}{2n}\sum_w w^2$, which is called *Elastic net regularization* [66].

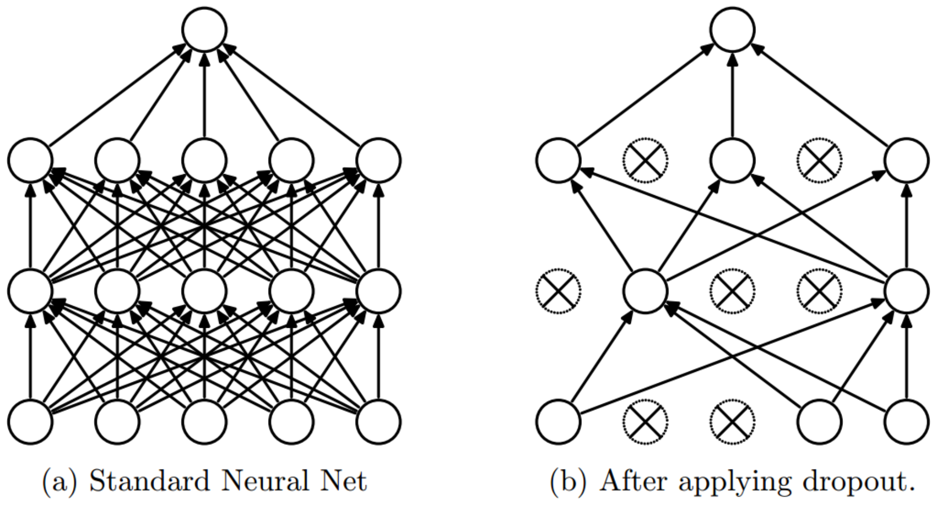


Figure 4.9: Dropout Demonstration. Source: [67].

Dropout is an extremely effective and simple regularization technique introduced by *Srivastava et al.* in [67]. Different from previous L1/L2 regularization methods, which is based on revising the objective function, dropout is realized by modifying the neural network itself. As shown in Figure 4.9, this technique will randomly drop units from the neural network during training, which prevents units from co-adapting too much [67]. During training, dropout can be interpreted as a sampled "*thinned*" neural network from the fully connected network, and only updating the parameters of the sampled network based on the input data. Thus, a neural network with n units could yield 2^n possible thinned neural networks. During testing, there is no dropout applied since it is hard to directly obtain the average of that exponentially many thinned models [67]. However, instead of applying dropout, a simple approximate averaging method is working very well, where a single neural network is used with each of the weight multiplied by the probability used during dropout of training. It is shown in [67] that dropout significantly improves the performance of neural networks on various supervised learning tasks in vision, speech recognition, document classification, etc.

Chapter 5

Design and Implementation

In this chapter, the system design for log template extraction, sequence generation and machine learning models are described. The very tricky issues of model training and evaluation techniques are analyzed. The system limitations are also provided for clarity.

5.1 Log Template Extractor

From above literature study and analysis of existing tools conducted in Chapter 3, it is making more sense to not consider the similarity at the character level but at the word (term) level. And we will make use of several ideas or notions together, and compare two to three methods mentioned above.

We define the string within a sentence between two adjacent delimiters as one *token* and the *Description* of one log entry is formed with multiple tokens, separated by the delimiters pre-defined. The *delimiters* are defined characters such as whitespace, symbols and certain punctuations (like `[`, `]`, `(`, `)`, `=`, `:`, `|`, etc.). This will bring the *Description* to a higher level of understanding, not in terms of single meaningless character but words. For better understanding our concepts regarding tokens, considering the log messages in Figure 5.1. For simplicity, the *Time-stamp*

```
osafimmnd[9146]: NO Ccb 595465 COMMITTED (BRFC)
osafimmnd[9146]: NO Ccb 595466 COMMITTED (BRFC)
osafimmnd[9147]: NO Ccb 595467 COMMITTED (BRFC)
```

Figure 5.1: Log examples.

and *Server-name* are ignored, and only the *Description* is provided. After being divided by the predefined delimiters, each of them contains seven tokens. For instance, the seven tokens of the first line of log entry are `osafimmnd`, `9146`, `NO`, `Ccb`, `595465`, `COMMITTED`, and `BRFC`. The template of these three lines of messages could be the Figure 5.2. where the tokens `9146` and `595465` are considered as **variables**

```
osafimmnd[*]: NO Ccb * COMMITTED (BRFC)
```

Figure 5.2: Log template example.

which are replaced be the wildcard symbol * in the template, and `osafimmnd`, `NO`, `Ccb`, `COMMITTED` and `BRFC` are the **constant** tokens. And the most possible corresponding Java code for printing our such logs are likely to be the following:

```
System.out.printf("osafimmnd[%d]: NO Ccb %d COMMITTED
    (BRFC)", pid, num);
```

In [7], Kimura *et al.* offer an optional way of how to define different word class as shown in Table 5.1. The higher the class number is, the higher its tendency to belong to the template is. For different requirements of the abstraction and

class	definition	examples
1	only numbers or numbers and symbols	1, 0/0, 10.1.1.1
2	numbers and letters	host-01, IPv4, L2TP, vty0, Fa0/0
3	symbols and letters	class-a, udp-port, aaa.cfg, line-protocol
4	only letters	linkdown, state, interface
5	only symbols	<, >, =, :

Table 5.1: Classes of Words defined in [7]

granularity level, the detailed definition of variables and constants will be varying. Since the log files are generated from diverse sources, the structures of them are heterogeneous (see Tables 2.2, 2.1 and 2.3) and it is also unwise to set regular expression for each of them for extracting their *Time-stamp*, *Description* and/or *Severity*, *Server-name*, etc. However, none of above methods are designed for such heterogeneous log packages. The machine learning algorithms will be applied to classify them (the categorical attributes of each of the log messages) into different classes (i.e., *Time-stamp*, *Severity*, *Server-name* or just free text of *Descriptions*).

As discussed in previous section, there are logs like the examples shown in Figure 2.4, where a fixed group of different templates of messages gathering together as a whole expressing one event. For such logs, each of the single log template is less interesting to us however one log entry is making more sense to be considered and abstracted as a whole for further log utilization and evaluation. The relevant examples are logs with multi-lines, logs represented as tables, etc.

The log template extractor is designed in two major stages: *log clustering*, *template extraction* and *sequence generation*. Generally speaking, the log clustering is process where similar logs are grouped together based on their editing distance. After the log clusters having been obtained, every single log template is extracted from each of the clusters gathered and then labeled by a unique identification for further usage. Now, a search dictionary is created as a key-value mapping, where the values are the log templates and the keys are the labels assigned to each of the templates. When a new set of log files are needed to be analyzed, each line of the logs will be fed into the search dictionary looking for corresponding label. Finally, a new sequence of labels is generated based on the new logs, where every ID is matching one log entry. In the following of this section, every step is discussed in detail.

Pre-processing is the step where the token types are identified and the less relevant tokens are tagged as wildcard symbol *. In this case, the less relevant tokens are the ones which are considered with lower possibility to be a part of the final template, such as numeric strings (integer, real number, float, hex .....), IPv4, IPv6, PCI address, ... On the contrary, they are more likely to be the variables. Also the logs like what Figure 2.4 is showing will be merged together treated as a single log entry sharing the same time stamp.

Tokenization is the step when every log entry string is divided based on the pre-defined delimiters. The delimiters are given, as regular expression in Python, in the Figure 5.3.

```
r'([\*\s,:()\[\]=|/\\{}\'\"<>\.\_\-])'
```

Figure 5.3: Delimiters.

Most of the delimiters are given based on the observations of the log data and the experiences. For example, the white space ‘ ’ is given naturally as a delimiter as it is taken as granted as a separator in natural language as well. The chars like colon ‘`:`’, different quotations ‘‘’"’, comma ‘`,`’, dot ‘`.`’, vertical line ‘`|`’ and various kinds of brackets ‘`[]()<>`’ are considered as delimiters also for the reason that they are more naturally being a token divider. The equal mark ‘`=`’ is regarded as a delimiter since in many logs, such as Figure 2.2, 2.5 and 2.6, the expression like `variable=value` appears, where the left side of the equal sign is the name of the variable and the right side is the corresponding value. The slash ‘`/`’ is commonly used within the file or command path. The underscore ‘`_`’ and dash sign ‘`-`’ is widely used in the new variable, instance or component naming.

One thing to remember is that the more relevant chars included as the delimiters, the more granular the tokenization would be. The reason is that it is tending to divide “*big*” tokens into “*small*” ones. This is having great impact on the following log clustering process since the more tokens there are within one log message, the more delicate the comparison it could be when detecting the editing similarities between two log messages simply because there are more tokens to be considered. However, it is over-considered if every char is regarded as a single token. This will greatly increase the computational time during similarity detection. Additionally, there is no need for judging the log similarity on the basis of each char when most of the tokens are consisted of fixed combination of chars.

Partitioning is a very helpful step, before the real log clustering, partitioning the original logs into several partitions based on their command type (i.e., the name of the process or component) and the length of each log (i.e., the number of tokens contained in the line of log).

It will dramatically reduce the computational time, especially plenty of time spent on detecting the Levenshtein distance and following log clustering. If you would like to find out a certain type of template among all templates, previously, the new log will be compared with all templates from every log cluster. However, if the middle class of hierarchical partitions are added, the system will first check which partition the new log belongs to based on the process name or component name of the log and the log length in terms of tokens. After the partition is identified, the log will be searched only within the clusters of this partition.

Generally speaking, we could just consider that clustering logs starting with different command (process) names into different clusters is naturally reasonable. The reason is that, intrinsically, the logs generated from the different process is normally having and should be considered with different log template. In this process, the hash table created is the mapping from the tuple `(command, length)` as the key to another hash table , where the logs are sharing the same command and log length. This hash table is described in the following. Therefore, there are actually two levels of hash tables, which is shown in Figure 5.4.

Log clustering is the process, given the new log message, after it has been assigned to a certain partition, it will be compared with every clusters in this partition and be grouped into the cluster which is having the minimal editing distance to the new log. If the minimal editing distance is less than a pre-defined threshold, the new log will be clustered into the corresponding group; otherwise, a new cluster will be created and the new log will be the representative log of this cluster. The problem of how to set a proper threshold value will be discussed in the *Chapter 6: Experiments and Results*, where the threshold is analyzed based on real results. In this process, the hash table created is actually the mapping from ID to a set of logs, which is similar to each other. The IDs are given as integers starting from `1, 2, 3, ...`, and whenever a new cluster is created, a new continous integer is given. Besides this, the integer `0` is kept for labeling unseen logs in the future.

In this project, the editing distance is based on the Levenshtein algorithms, which could be mathematically given as the Equation 5.1.

$$\mathrm{Lev}_{\mathrm{A,B}}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j)=0 \\ \min \begin{cases} \mathrm{Lev}_{\mathrm{A,B}}(i-1,j)+1 \\ \mathrm{Lev}_{\mathrm{A,B}}(i,j-1)+1 \\ \mathrm{Lev}_{\mathrm{A,B}}(i-1,j-1)+1_{(\mathrm{A}_i \neq \mathrm{B}_j)} \end{cases} & \text{otherwise} \end{cases} \tag{5.1}$$

As discussed earlier in this report, the Levenshtein editing distance $\mathrm{Lev}_{\mathrm{A,B}}(|\mathrm{A}|,|\mathrm{B}|)$ is the minimal actions it will take to change string A to string B in terms of the

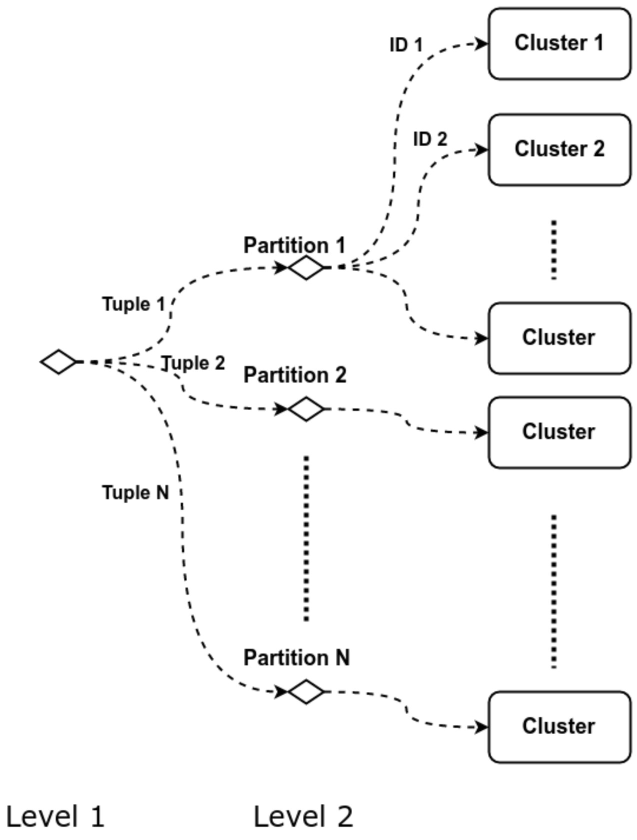


Figure 5.4: Two-level hash table.

insertions, deletions, substitutions (and transpositions). In our case, instead of actions on char level, the editing actions are on the basis of word tokens. It can be solved in many difference ways such as recursion, iteration with full matrix or iteration with two simple vectors. Considering the efficiency of the algorithm, the existing tool called `editdistance` [68] is utilized in this thesis project, which is originally written in C++ and based on the work of [69]. According to the benchmark of [68], `editdistance` is outperforming most of other tools and also supporting token-based detection besides of char-based one.

Template extraction is just the step where a single log template representation is extracted from each of the log clusters. The way of doing this is considering the cardinality of the tokens (number of unique tokens) at each position of the log with certain length. Based on the previous partitioning step, it is sure that the logs within a same cluster is having the same length. By counting the unique tokens at each position of the logs within a cluster, it is easy to just simply replace the content with the position, where more than one unique tokens are appearing, by wildcard symbol *. Finally, the template representation for this cluster is generated with original unique tokens and replaced wildcard symbols. For example, in Figure 5.5, the positions of A, B and D are having more than one unique tokens whereas the C, E, F are having only one unique token. Then the template obtained is `* * C * E F G`.

```
A1 B1 C D2 E F G
A2 B1 C D1 E F G
A1 B2 C D1 E F G
```

Figure 5.5: Delimiters.

In this process, the previous hash table, mapping IDs to the log set, is transfered to the mapping from IDs to their corresponding log template.

Search dictionary generation is the process, during the log clustering and template extraction, where a search dictionary is also created for matching the new logs and labeling them in the future. It is a hierarchical hash table, where there are two mappings. The first mapping is from the tuple key `(command, length)` to the specific partition, which is another hash table. This hash table is the second level of the hierarchy, where the key is the ID labeled on each of the templates and the value is the corresponding template representation. The purpose of the dictionary is that, when a new log is fed into the system, the program will match the log with the most similar template in the system and output this template's ID for the following sequence generation. The new log will be firstly mapped to a partition then to the specific log template.

Log matching is the process, where new logs are matched with the obtained template representation stored in the search dictionary for obtaining their own IDs. The log matching is actually divided into two steps. First, the command token of the new log extracted together with the log length in order to create a tuple `(command, length)`. Using this tuple as the key, a partition of templates referring different clusters will be found for further matching. Then, the log will be compared with each of the template representations for similarity check. If there is a match, the new log will be given with the corresponding key ID of the value template; otherwise, the ID `0` will be given meaning that this is an *unknown* log. The log matching process is demonstrated in the Figure 5.6.

Sequence generation is the process, where a new sequence of labels (IDs) is generated based on the new set of logs. This is done by previously discussed log matching function based on the search dictionary created. The generated ID sequences are regarded as the input to the machine learning models for sequence learning.

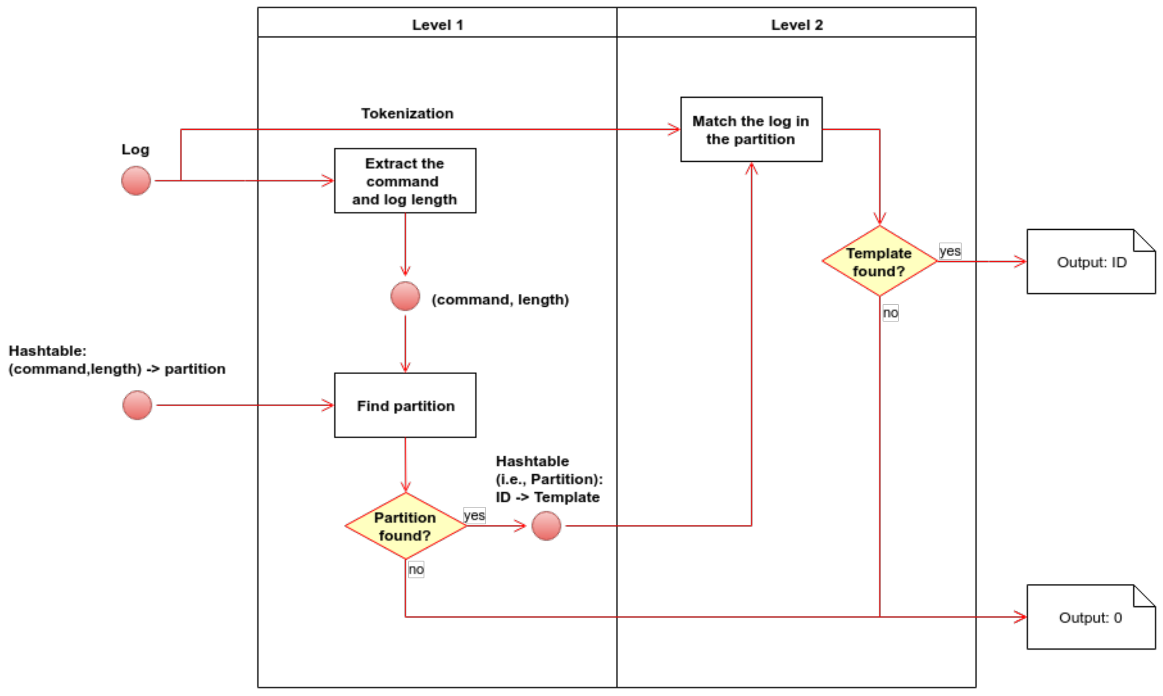

Level 1
Level 2
Tokenization
Log
Extract the command and log length
Match the log in the partition
Template found?
yes
no
Output: ID
(command, length)
Hashtable: (command,length) -> partition
Find partition
Hashtable (i.e., Partition): ID -> Template
Partition found?
yes
no
Output: 0

Figure 5.6: Log Matching Process.

5.2 Naive Bayes Model

First of all, several remarks and definitions are given for better explanation of the machine learning problem and demonstration of the training and validation data set.

Remark 5.2.1. K - the maximal class label and all class labels are given as integers equal or less than `K`.

Remark 5.2.2. `ID n` - the nth ID on the sequence `ID 1, ID 2, ID 3, ..., ID N` with length of N, where $n \in \{1, 2, ..., N\}$. The `ID n` is belonging to the set of ID classes `{0, 1, 2, ..., K}`,

Remark 5.2.3. *sentence* - the subset of a sequence. It is used to train the model as the input data of each of the single data sample. For a sequence with length of N, we defined M as the length of sentence, where M is less than N. The length of sentence is also called as window size.

Remark 5.2.4. $X\_train \in \mathbb{Z}^{S \times M}$ - the input data in the training data set for Naive Bayes model. It is with the dimension of $[S, M]$, where S is the number of samples in the training data set and M is the length of sentence, i.e., the window size. In other words, X_train is a set of sentences.

Remark 5.2.5. $y\_train \in \mathbb{Z}^{S}$ - the output data in the training data set for Naive Bayes model. It is a S-length vector, where S is the number of samples in the training data set and S is also equal to the number of samples in X_train. In other words, each data of y_train, as the target data, is matching each input data of X_train.

The X_train and y_train are shown in Figure 5.7. In this project, the problem

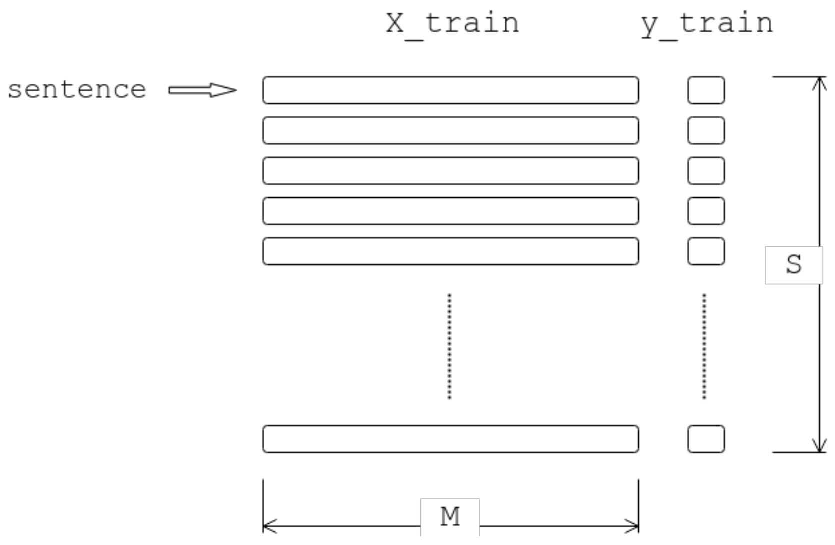


Figure 5.7: X_train and y_train.

of sequence learning and sequence detection is demonstrated in Figure 5.8. Given

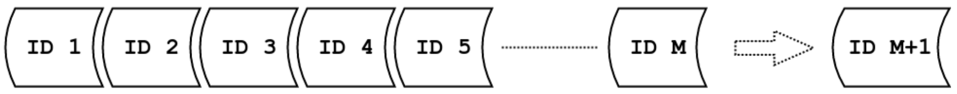


Figure 5.8: Sequence Prediction Demonstration.

an ID sentence `ID 1, ID 2, ID 3, ..., ID M`, the machine learning model, after training, should be able to predict the next correct ID, i.e., `ID M+1`. The number M is defined as window size in Remark 5.2.3. The problem, in other words, is actually the problem of calculating what is the ID among K classes with highest probability at the position of `ID M+1`, given its previous M IDs are `ID 1, ID 2, ID 3, ..., ID M`. Applying the Naive Bayes theory described in Section 4.1, the following Equation 5.2 is obtained by rewriting the Equation 4.7.

$$\begin{aligned} &P\Big(\mathrm{ID}_{M+1} = \mathrm{id}_k \;\Big|\; \mathrm{ID}_1 = \mathrm{id}_{i_1}, \mathrm{ID}_2 = \mathrm{id}_{i_2}, \dots \mathrm{ID}_M = \mathrm{id}_{i_M}\Big) \\ \propto\; &P(\mathrm{ID}_{M+1} = \mathrm{id}_k) P\Big(\mathrm{ID}_1 = \mathrm{id}_{i_1}, \mathrm{ID}_2 = \mathrm{id}_{i_2}, \dots \mathrm{ID}_M = \mathrm{id}_{i_M} \;\Big|\; \mathrm{ID}_{M+1} = \mathrm{id}_k\Big) \\ \propto\; &P(\mathrm{ID}_{M+1} = \mathrm{id}_k) \prod_{j=1}^{M} P\Big(\mathrm{ID}_j = \mathrm{id}_{i_j} \;\Big|\; \mathrm{ID}_{M+1} = \mathrm{id}_k\Big) \end{aligned} \tag{5.2}$$

where

$$k, i_j \in \{1, 2, ..., K\}.$$

The task now is to obtain the id_k for $k \in \{1, 2, ..., K\}$ so that the value of Equation 5.2 will be maximized. In order to obtain the maximal value of posterior as Equation 5.2, the values of prior $P(\mathrm{ID}_{M+1} = \mathrm{id}_k)$ and likelihood $P\Big(\mathrm{ID}_j = \mathrm{id}_{i_j} \Bigm| \mathrm{ID}_{M+1} = \mathrm{id}_k\Big)$ should be obtained, for $k, i_j \in \{1, 2, ..., K\}$ and $j \in \{1, 2, ..., M\}$.

The major class `NaiveBayes` is defined. There are two major steps of Naive Bayes model: *learning* and *evaluation*. And the details of both of them will be discussed and illustrated in the following of this section. The full code including model building, training and evaluating can be found in my Github repository: `https://github.com/fluency03/sequence-rnn-py/blob/master/naive_bayes.py`.

5.2.1 Training

In the learning process, two matrices are created for computing the above probabilities: (i) $ny \in \mathbb{Z}^K$ is the vector representing the number of appearance for each different class in y_train, where K is the number of classes; (ii) $nx\_y \in \mathbb{Z}^{M \times K \times K}$ is the matrix representing the number of different classes appearing in X_train, at each position of $m \in \{1, 2, ..., M\}$, given certain data in y_train. The values of the two matrices are initialized with zeros and added up during the training process, which will be used in the evaluation process for calculating the probability distributions.

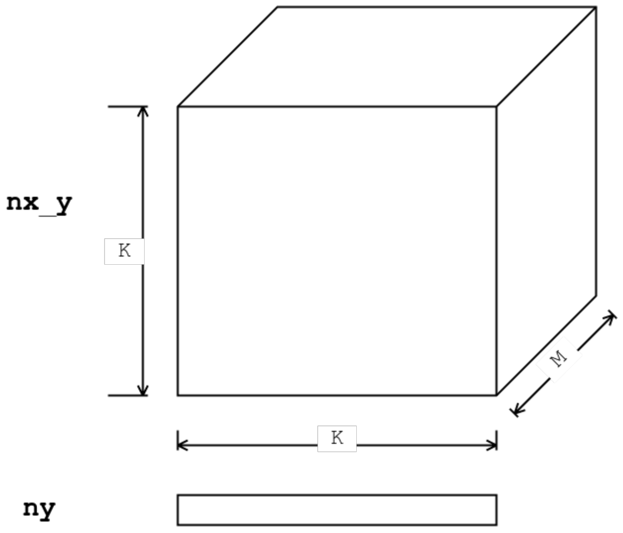


Figure 5.9: ny and nx_y.

The calculation of these two matrices is shown in Listing 1 in Python code. Given the training data sets `X_train` and `y_train`, the number of samples `S` is obtained first. For each of the sample `i` in `S`, the value of `ny[y_train[i]]` is increased

```
# ------------------- Likelihood ------------------- #
px_y = np.zeros((self.nb_classes, self.window_size))
for p in xrange(self.nb_classes):
    for k in xrange(self.window_size):
        px_y[p, k] = ((self.nx_y[k, X_eval[i, k], p] +
                       self.alpha) /
                      (self.ny[p] +
                       self.alpha * self.nb_classes))
```

Listing 1: Train Naive Bayes Model.

by 1 when the certain ID class `y_train[i]` appears once. For each of the target value `y_train[i]`, the matrix `nx_y` is calculated based on every position `i` within the `window_size` which is M mentioned in previous section. The value of `nx_y[j, X_train[i, j], y_train[i]]` is increase by 1 when the ID `X_train[i, j]` is discovered at position `j` within window of size - `window_size`, for the corresponding target data `y_train[i]`. `X_train[i, j]` is the ID at the position `j` of the sentence `i` within the input data set `X_train`.

5.2.2 Evaluation

In the evaluation process, based on the two numeric matrices obtained previously, three relevant probability matrices can be obtained: (i) the prior $py \in \mathbb{R}^K$, (ii) the likelihood $px\_y \in \mathbb{R}^{K \times M}$, and (iii) the posterior $py\_x \in \mathbb{R}^K$. The probabilities are calculated for each of the data sample within the evaluation data set `(X_eval, y_eval)`. `X_eval` and `y_eval` are sharing the same space dimension with `X_train` and `y_train` mentioned before, respectively, but with different number of data samples.

Additionally, a value $\alpha \geq 0$ is defined as the smoothing prior, which is used for taking features not present in the learning samples into account. It is capable of preventing zero probabilities so that, in future computation, no previous zero probabilities could lead to a final zero probability after multiplied together. When $\alpha = 1$, the method is called *Laplace smoothing*; while $\alpha < 1$, it is called *Lidstone smoothing* [70]. The usage of α is demonstrated in the following part.

The Python code in Listing 2 shows the process of prior calculation. The prior

```
total_y = np.sum(self.ny)
# --------------------- Prior --------------------- #
py = np.zeros(self.nb_classes)
for c in xrange(self.nb_classes):
    py[c] = ((self.ny[c] + self.alpha) /
             (total_y + self.alpha * self.nb_classes))
```

Listing 2: Evaluate the Naive Bayes Model - Prior Calculation.

can be represented as Equation 5.3 for each ID class $c \in \{1, 2, ..., K\}$, where the smoothing parameter α is utilized. In this equation, without α, if the number of certain class in target data, i.e. `ny[c]`, is equal to zero, the probability `py[c]` will be zero as well. However, it is really unfair to say the chance of certain data's appearance is zero just because it is not in the training data. Therefore, a small number is added on this probability even when `ny[c]` is zero in the training data.

$$\mathrm{py}[c] = \frac{\mathrm{ny}[c] + \alpha}{\mathrm{total\_y} + \alpha * \mathrm{nb\_classes}} \tag{5.3}$$

In the following part of this section, the probabilities (likelihood and posterior) are calculated based on the sample `i` for each of data pair `(X_eval[i], y_eval[i])`. The Python code in Listing 3 shows the process of likelihood calculation. Given

```
# ------------------- Likelihood ------------------- #
px_y = np.zeros((self.nb_classes, self.window_size))
for p in xrange(self.nb_classes):
    for k in xrange(self.window_size):
        px_y[p, k] = ((self.nx_y[k, X_eval[i, k], p] +
                       self.alpha) /
                      (self.ny[p] +
                       self.alpha * self.nb_classes))
```

Listing 3: Evaluate the Naive Bayes Model - Likelihood Calculation.

certain data `p` in target output `y`, the probability of evaluation data `X_eval[i, k]` at position `k` on the sentence `X_eval[i]` with length `window_size` is represented in Equation 5.4.

$$\mathrm{px\_y}[\mathrm{p}, \mathrm{k}] = \frac{\mathrm{nx\_y}\Big[\mathrm{k}, \mathrm{X\_eval}[\mathrm{i}, \mathrm{k}], \mathrm{p}\Big] + \alpha}{\mathrm{ny}[\mathrm{p}] + \alpha * \mathrm{nb\_classes}} \tag{5.4}$$

The Python code in Listing 4 shows the process of posterior calculation. In

```
# ------------------- Posterior ------------------- #
py_x = np.zeros(self.nb_classes)
for j in xrange(self.nb_classes):
    py_x[j] = py[j] * np.prod(px_y[j])
```

Listing 4: Evaluate the Naive Bayes Model - Posterior Calculation.

Equation 5.5, the probability of class `j` given the evaluation data `X_eval[i]` of sample `i` is calculated, based on the prior `py[j]` and the likelihood - the product of each probability `px_y[j, k]` at each position `k` within the window.

$$\mathrm{py\_x}[\mathrm{j}] = \mathrm{py}[\mathrm{j}] * \prod_{k=1}^{M} \mathrm{px\_y}[\mathrm{j}, \mathrm{k}] \tag{5.5}$$

For each data sample `i`, the predicted class is the one with highest probability within the posterior `py_x`. And the ground truth (target value) is the `y[i]` for sample `i`. This is shown as Python code in the Listing 5. The number of correct predictions `nb_correct` is initialized as zero and adds `1` when the prediction is correct for sample `i`, i.e., when `y_true == y_pred`.

```python
# ------------------- Prediction ------------------- #
y_pred = np.argmax(py_x)
y_true = y_eval[i]
if y_true == y_pred:
    nb_correct += 1
```

Listing 5: Evaluate the Naive Bayes Model - Prediction.

Finally, the accuracy is calculated from the number of correctnesses over the overall predictions, i.e., `accuracy = nb_correct / len(y_eval)`.

Regarding above computation process, a place that can be improved is remaining in the Equation 5.5 and Listing 4. It is noticed that logarithms is not applied in Equation 5.5 and, consequently, when there are many small numbers (less than 1.0) multiplied together, the product of them will become extremely small, which could lead to the underflow problem. That means when the product is smaller than the bottom range of the float number in Python, the result will become zero. The logarithms of posterior `py_x` is given in Equation 5.6.

$$\log\Big(\mathrm{py\_x[j]}\Big) = \log\Big(\mathrm{py[j]}\Big) + \sum_{k=1}^{M} \log\Big(\mathrm{px\_y[j,k]}\Big) \tag{5.6}$$

However, for this application it did not make any difference in the results, so Equation 5.5 was used throughout. To be absolutely certain, I have tested at least one NB case with both non-logarithms and logarithms just to make sure I get the same result. [1]

5.3 Recurrent Neural Networks

In this project, a deep learning library called *Keras* [71] is utilized for RNN modeling and evaluation. According to its official documents, Keras is *a minimalist, highly modular neural networks library, written in Python and capable of running on top of either TensorFlow or Theano.* Due to the design principles of high level modularity, minimalism and extensibility of Keras, it is very easy to quickly build up a neural network prototype. Because of running on Theano (or TensorFlow), Keras is also seamlessly supporting CPU and GPU. Thanks to the flexibility of network modeling, arbitrary connectivity scheme is supported, such as multi-input/multi-output

[1]Please refer to the *Chapter 6. Experiments and Results.*

training, sequential/graph model, convolutional and recurrent combined networks, etc.

In this project work, the Keras RNN model is trained on the basis of Theano and the GPU is also enabled. The simple `Sequential` model is chosen for the basic structure instead of `Graph` model. The full code of Recurrent Neural Network including model building, training and evaluating can be found in my Github repository: `https://github.com/fluency03/sequence-rnn-py/blob/master/rnn_sequence_analyzer.py`.

The data sets used for RNN training and validation are slightly different from the ones used for Naive Bayes model. Based on Remarks 5.2.4 - 5.2.4, the data sets are modified for RNN in the following remarks.

Remark 5.3.1. $X\_train \in \mathbb{Z}^{S \times M \times K}$ - the input data in the training data set for RNN model. The difference of X_train for RNN from the one for Naive Bayes model is that X_train for RNN has one more dimension in space - the number of class K. In Naive Bayes, each ID is represented by a single integer; however, in RNN, each ID is represented by a *one-hot vector*.

Remark 5.3.2. *one-hot vector* - the vector with length of K, which is equal to the number of classes. In such vector, only one value is `1` and all the rest are `0`s. For the integer n, it is represented by the K-vector, where the n-th value is `1` and the rest of them are `0`s.

Remark 5.3.3. $y\_train \in \mathbb{Z}^{S \times K}$ - the output data in the training data set for RNN model. It has the same difference mentioned in Remark 5.3.1, i.e., it has one more dimension - the number of classes K.

5.3.1 Model and Layers

In Keras, the `Sequential` model can be simply initialed as `model = Sequential()`. The LSTM layer is given in Listing 6, where most of the parameters are ignored. Please refer to Keras's official document [72] for detailed information. The param-

```
keras.layers.recurrent.LSTM(output_dim, ...)
```

Listing 6: LSTM Layer in Keras.

eter `output_dim` indicates the dimension of the internal projections (i.e., the length the layer when it is a hidden layer) and the final output (i.e., the length the layer when it is the output layer).

The class `LSTM` is inherited from the abstract class `Recurrent` shown in Listing 7. The paramter `weights` is the list of arrays to manually set the initial weights and the default values are initialized as zeros when it is `None`. The `return_sequences` is a `Boolean` value, which indicates whether to return the last output in the output sequence, or the full sequence. The `go_backwards` (a `Boolean` value, default as `False`) indicates whether to process the input sequence backwards, which is very

```
keras.layers.recurrent.Recurrent(weights=None,
                                 return_sequences=False,
                                 go_backwards=False,
                                 stateful=False,
                                 ...
                                 input_dim=None,
                                 input_length=None)
```

Listing 7: Recurrent Layer in Keras.

helpful in Bi-directional RNN model. The `stateful`: Boolean (default False). If True, the last state for each sample at index i in a batch will be used as initial state for the sample of index i in the following batch. Additionally, two parameters `input_dim` and `input_length` define the dimension of the input data. They are used only when th layer is the first layer of the model, and in practice, the compact argument as `input_shape=(input_length, input_dim)` is applied. The `input_dim` is the dimensionality of the input data, which is the length of one-hot vector, i.e., the number of classes. The `input_length` is the length of input sequences, i.e., the length of one single sentence. It is the number of time series steps, which the model will be based on for recurrently processing the data. There are also other parameters but kept non-explained in this report.

As an abstract class, `Recurrent` itself cannot be used to form any valid layer in a model. And the child class `LSTM` (along with other children like `GRU` and `SimpleRNN`) inherits the properties from `Recurrent`, such as the parameters, members and methods.

The Listing 8 shows how the dropout mechanism is applied, where the argument `p`, as a float value ranging between $[0, 1]$, indicates the fraction of the dropout between the output of previous layer to the input of next layer. It is a powerful regularization method and mainly used for avoiding overfitting.

```
keras.layers.core.Dropout(p)
```

Listing 8: Dropout Layer.

The Listing 9 shows how the activation function is applied to a layer, where the argument `activation` is the function type, represented as string.

```
keras.layers.core.Activation(activation)
```

Listing 9: Activation Layer.

In Keras, the class `TimeDistributed` is defined as a layer wrapper, where the input argument is a class of layer. When a layer is applied into the wrapper

TimeDistributed, the dense connection is based on the unit of each of the time steps. Two sequence training schemes can be applied: *many to one* and *many to*

```
keras.layers.wrappers.TimeDistributed(layer)
```

Listing 10: Time Distributed Layer Wrapper.

many, which are mentioned in Figure 4.6.

5.3.2 Many-to-one and Many-to-many

In order to demonstrate these two approaches, a training sequence is given as following: ID 1, ID 2, ID 3, ..., ID N. In many-to-one case, the above sequence is divided into the following set of data samples for training in Figure 5.10, where k

```
        X_train                                          y_train
  ID 1, ID 2, ..., ID k                                   ID k+1
  ID 2, ID 3, ..., ID k+1              predict ->         ID k+2
  ID 3, ID 4, ..., ID k+2                                 ID k+3
  ...
  ID N-k, ID N-k+1, ..., ID N-1                           ID N
```

Figure 5.10: Many to One.

is the length of sentence related to the window size mentioned in Naive Bayes. The length of each data sentence limits how much the gradients can propagate backwards in time. For example, if sentence length is 20, then the gradient signal will never back-propagate more than 20 time steps, and the model might not find dependencies beyond this length of IDs. This is actually the limitation of the model's long term memory. Thus, if difficulty relies on the dataset where there are a lot of long-term dependencies, increasing the sentence length might be a proper choice.

For the many-to-many approach, the target data y_train is a subsequence instead of a single ID shown in Figure 5.11. In this case, each sample in y_train is

```
       X_train                                          y_train
ID 1, ID 2, ..., ID k                           ID 2, ID 3, ..., ID k+1
ID 2, ID 3, ..., ID k+1                         ID 3, ID 4, ..., ID k+2
ID 3, ID 4, ..., ID k+2          predict ->     ID 4, ID 5, ..., ID k+3
...
ID N-k, ID N-k+1, ..., ID N-1                ID N-k+1, ID N-k+2, ..., ID N
```

Figure 5.11: Many to Many.

the exact next sentence to its corresponding data sample in X_train.

- many to one - This approach is very specifically designed to learn the exact next ID after certain subsequence. Other common applications, in this case, would be the classification problems, where the features near to each other are more independent. However, in the sequence learning and prediction problem, the IDs near to each other are very closely related.

- many to many - This approach is more capable of learning the dominant patterns among the whole sequence instead of only predicting the next ID.

In the many-to-one problems, the loss is calculated only based on the error (level of difference) of the predicted next ID and ground truth. However, in the many-to-many problems, the loss is calculated on each of the IDs among the sequence. That means, given the sentence sample of the targeted training data `y_train` with length of L, the total loss is obtained based on each of the L IDs. More targeted data points needed to be fit makes the model weights harder to adjust, which leads to a slower training process. On the contrast, it is a relatively reasonable choice in the sequence learning and prediction problem, where there could be long term dependencies.

Additionally, the many-to-many approach can be extended to unfixed, unequal sentences' lengths, both in input side and output side. The relevant applications are the encoder-decoder or sequence to sequence learning. For example, the *addition* application, where the input sequence is "`535+61`" with length 6 and the output sequence is "`596`" with length 3. The Keras-based python code of application *addition* is given in the Github repository: `https://github.com/fchollet/keras/blob/master/examples/addition_rnn.py`.

5.3.3 Build the Model

In Keras, constructing a neural network model is as easy as the method `add` is called by the `model` for each of the layer. This brings the problem about the size of the model:

- How many recurrent layers should be added to the model?

- How many units should each layer contain?

This is a very tricky problem to every neural network case. Many things should be considered and experimented in order to determine the size of the network. However, by conducting the approximate calculation of the total among of model's parameters and comparing it to the total data size, a general feeling of the model size can be obtained. The following calculation inspired from [73] offers a simple example about how to compute the number of parameters within a model.

Consider one layer of LSTM units, if it has the layer size of H=512 and if we have the vocabulary size as C=3000 (the number of unique classes, i.e., the number of unique IDs in our case), the LSTM layer will have three parameter matrix:

- U with dimension (H, C)=(512, 3000);
- V with dimension (C, H)=(3000, 512);
- W with dimension (H, H)=(512, 512).

Then, the total number of parameter for one layer will be $2HC + H^2$, which is $3,334,144$ in this case. That is more than 3 million parameters for only one layer! As indicated in [73], the approximate number of parameters (i.e., the model size) in a neural network can be adjusted based on the amount of data. For example, for a dataset with 100 million IDs (which represents 100 million logs), a model with only 100 thousands parameters should be obviously insufficient. An underfitting situation is very likely. On the contrast, if there are only 100 thousands IDs in the dataset but the model is with size of millions of parameters, the overfitting is also hardly avoidable. In addition, if it is a 1-million dataset running on a 1-million-parameter model, it is still necessary to carefully monitor the training loss and validation loss in order to avoid overfitting. One commonly used trick is to increase the model size meanwhile set a higher dropout rate.

In a word, according to [73], these two factors, i.e., the number of parameters and the amount of data, should be about the same order of magnitude. It is also suggested that the number of layers will normally be 2 or 3.

Finally, in order to compile and build a neural network model, the function `compile` defined in Listing 11 is called as following: `model.compile(loss='categorical_crossentropy', optimizer='rmsprop')`. In the `compile` method, the parameter

```
compile(self, optimizer, loss, metrics=[], sample_weight_mode=None)
```

Listing 11: Compile the Keras model.

`optimizer` indicates the different schemes of gradient descent optimization, such as stochastic gradient descent (SGD), RMSprop, Adagrad, Adadelta, etc. For RNN model, `rmsprop` is a good choice as recommended in Keras's official document. It is defined in Listing 12, where an important neural network influence factor learning rate `lr` is applied. The `loss` represents the loss function utilized, which are men-

```
  keras.optimizers.RMSprop(lr=0.001, rho=0.9, epsilon=1e-08)
```

Listing 12: RMSprop Optimizer.

tioned in Section 4.2.3. For our sequence prediction problem (which actually is the problem of next ID classification), the `categorical_crossentropy` is chosen. The parameter `metrics` is used for model evaluation during training and validation. For example, the accuracy can be recorded by applying `metrics=['accuracy']`.

5.3.4 Training and Validation

The factor `lr` (learning rate) plays a great role on the speed (step of the gradient descent) and quality of learning. The higher the learning rate is, the faster the neuron trains. The lower it is, the more accurate the training is, especially when the training process tends to converge. For example, if the learning rate is very large, the neural network will not be able to converge to the a more optimized, minimal place, because each step is too big, even though there are more training iterations. According to [65], *the learning rate is by far the most important hyperparameter*. The tricky part of learning rate is that it is very hard to select a proper value of it, which differs from various amount and variance of data set, diverse usage cases, etc. It is suggested that, in practice, it is sufficient to do a rough search by starting with a high value (e.g. 1.0) and dividing it by ten every time until the performance stops increasing or tends to converge [65].

In order to train the RNN model defined, compiled and built in previous section, the method `fit` will be called by `model`. The method is given in Listing 13.

```
fit(self, x, y, batch_size=32, nb_epoch=10, verbose=1,
    callbacks=[], validation_split=0.0, validation_data=None,
    shuffle=True, class_weight=None, sample_weight=None)
```

Listing 13: Train the RNN model.

The parameters `x` and `y` are the input data for model training. The parameter `batch_size` indicates the number of samples per gradient update during training and gradient descent process. The larger the batch size is, the more data can be trained in parallel. To some extent, this action could accelerate the training speed. However, when the batch size exceeds a limitation, the computational resources are also constrained by the amount of data so that the speeding-up will be saturated. The `nb_epoch` is the number of epochs (i.e., iterations) to train the model. The `verbose` decides whether the information will be printed out during training. The `validation_split` is the fraction of validation data among the whole data set used for model training. Normally, in practice, the fraction is around 5% to 10%. Additionally, the parameter `validation_data` can also be used to specifically indicates the data for model validation.

5.3.5 Evaluation

For the evaluation of the model given a new set of data, the following method `predict` in Listing 14 is called in order to obtain the neural network prediction, which will be used to compare with the ground truth value. Similar arguments applied in method `fit` are also given for method `predict`.

In the evaluation process, the number of correct predictions `nb_correct` is initialized as zero. Whenever the model predicted value `y_next_pred` is equal to the

```
predict(self, x, batch_size=32, verbose=0)
```

Listing 14: Evaluate the RNN model.

ground truth value `y_next_true`, `nb_correct` is increase by one. After all predictions are finished, the accuracy is obtained as: `nb_correct / nb_samples`, where `nb_samples` is the number of samples in the evaluation data set.

Additionally, from the `predict` method, the probability of each of the ID class can be obtained as well. Therefore, the chance of the ground truth value for each of the data sample can be collected and plotted. Based on such results, this project also intend to find out the anomaly places, where the ground truth data are treated with very low probabilities. Ideally, this is the place, we believe, where the unexpected behavior of the system happens and the testers should pay more attention. The relevant results will be given in the next chapter.

Chapter 6

Experiments and Results

In this chapter, the results of Log Template Extractor are demonstrated and illustrated. Several experiments are conducted in order to obtain deeper understanding of the difficulties in RNN training. It also offers the reasons regarding certain decisions made during the RNN training process. Finally, the RNN model is used to attempt to solve the practical sequence prediction and anomaly detection problem, with comparison of the results from Naive Bayes model.

6.1 Log Template Extractor

The Log Template Extractor is hardly numerically evaluated since there is no right or wrong answer. There is no (pre-defined) targeted log clusters or specific template representation. For different original raw log data, it is also not certain that how many clusters will be formed and how many templates will be extracted from them. With regard of one time of template extraction process, it is also not determined which template is labeled by which ID integer.

In this section, I will present and analyze some statistics related to the log template clustering and extraction. These experiments and results will demonstrate how I made some decisions during this process.

Number of clusters vs. Number of logs.

#log files	1	2	3	4	5	6	7	8	9
#clusters	2906	2945	3057	3060	3100	3108	3127	3134	3156
#log files	10	12	14	16	18	20	24	28	32
#clusters	3161	3169	3183	3189	3208	3213	3302	3409	3546

Table 6.1: Number of clusters vs. Number of logs.

Table 6.1 shows the relationship between the number of clusters generated and number of log files imported. Each log file contains at least 43,000 logs, which means 32 log files have more than 1,376,000 logs in total. As it is shown in the table, one log

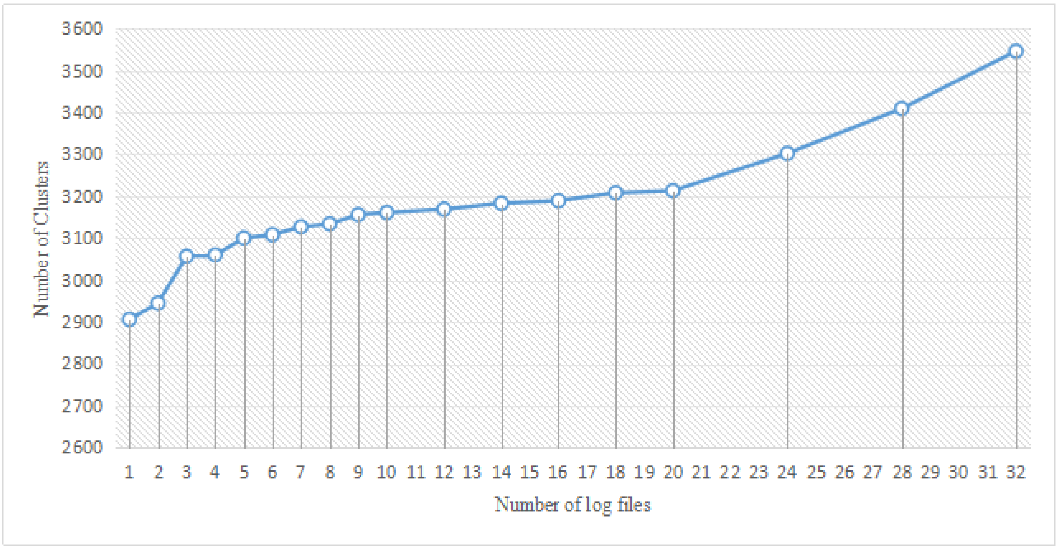


Figure 6.1: Number of clusters vs. Number of logs.

file with at least 43,000 logs will converge to 2906 clusters, each of which represents a log template where the logs belonging to this cluster are similar to each other. With the gradually increasing of the number of log files imported, the number of clusters created is increased accordingly. Most of the clusters are contained within each of the log files since there are more than 81% [1] clusters generated from all of the 32 log files can be generated from a single file. However, it is still no doubt that the more log files collected, the better the results could be since more previously unseen templates will be extracted as well as it is shown in Figure 6.1.

In the above experiment, the threshold is set to 0.1. The threshold represents the percentage of dissimilarity between two logs when they are compared and clustered. When a new log is compared with all the created cluster representations, if the obtained dissimilarity value is lower than or equal to the predefined threshold, that means this log could belong to the cluster. If more than one cluster are meeting this requirement, the new log will be grouped into the cluster with lowest result. If all results are higher than the threshold, a new cluster will be created and the new log will become the initial representation of this cluster. Now, another experiment is conducted to evaluate the effect of this preset threshold.

Number of clusters vs. Threshold.								
threshold	0.05	0.1	0.15	0.2	0.3	0.4	0.5	0.6
#clusters	4655	3546	3226	3042	2626	2251	1919	1652

Table 6.2: Number of clusters vs. Threshold.

[1] $2906 \div 3546 \times 100\% \approx 81.95\%$

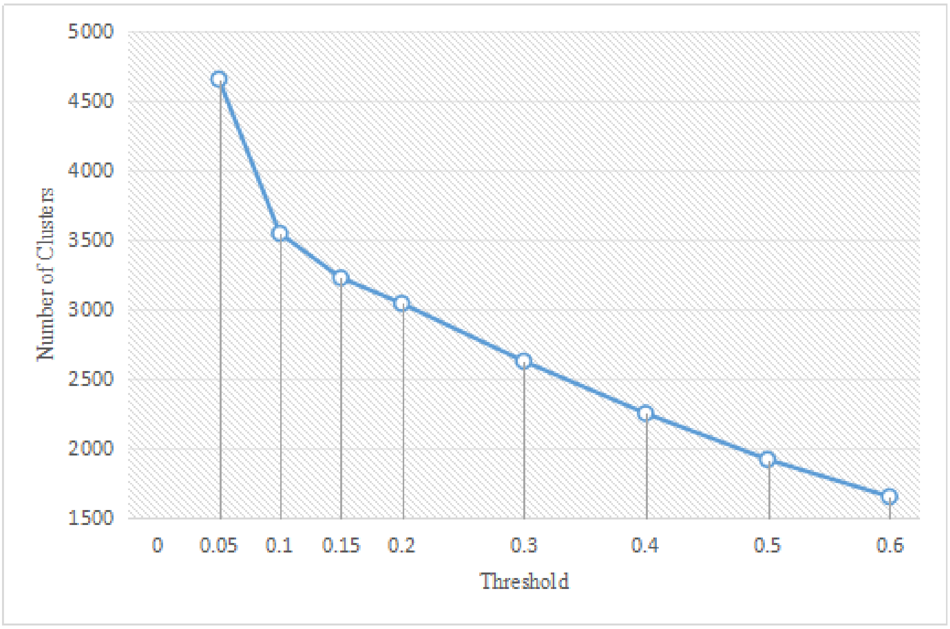


Figure 6.2: Number of clusters vs. Threshold.

Table 6.2 and Figure 6.2 show the relationship between the number of clusters that can be obtained and the threshold value. This experiment is based on 32 log files. With the increasing of the threshold value, the number of clusters created is decreased accordingly since the system is tolerating bigger difference between the logs within a cluster thus more logs will be grouped together into one cluster. Based on the numbers given in Table 6.2, on one hand, it is shown that there is a big difference from `threshold = 0.05` to `threshold = 0.1` [2] since, when threshold is equal to 0.05, it is too strict when detecting the difference and creating new clusters. For example, for two logs with maximum length of 20 tokens, `threshold = 0.05` means only one-token-difference between them is allowed otherwise they will be treated as different logs. This will generate too many classes (clusters) for machine learning tasks however most of these classes only appear one or two times. On the other hand, larger threshold means many logs, which should be treated differently, are gathered together into the same cluster. This will lead to the situation that many different logs are belonging to the same class, which is also bad to the following machine learning tasks since these logs are, unfortunately treated as same. Therefore, `threshold = 0.1` should be a reasonable choice.

Finally, some typical examples of found templates are given as following:

[2] $4655 - 3546 = 1109$

```
324
kernel: [    *.*] smpboot: Total of * processors activated (*.* BogoMIPS)
325
kernel: [    *.*] RTC time: *:*:*, date: */*/*
326
kernel: [    *.*] PCI: MMCONFIG at [mem *-*] reserved in *
327
kernel: [   *.*] ACPI: Executed * blocks of module-level executable AML code
328
kernel: [    *.*] pci *:*:*.*: System wakeup disabled by ACPI
329
kernel: [    *.*] pci *:*:*.*: PME# supported from * D0hot
330
kernel: [    *.*] pci_bus *:*: root bus resource [bus *]
331
kernel: [    *.*] system *:*: [mem *-*] has been reserved
332
kernel: [    *.*] pci_bus *:*: resource * [mem *-*]
334
kernel: [  *.*] pciehp: PCI Express Hot Plug Controller Driver version: *.*
335
kernel: [  *.*] ERST: Error Record Serialization Table (ERST) support is initialized.
336
kernel: [   *.*] PM: Hibernation image not present or could not * loaded.
337
kernel: [  *.*] shpchp: Standard Hot Plug PCI Controller Driver version: *.*
338
kernel: [   *.*] IPv0: ADDRCONF(NETDEV_UP): *: link is not ready
339
kernel: [   *.*] IPv0: ADDRCONF(NETDEV_CHANGE): ***: link becomes ready
340
kernel: [   *.*] block drbd0: drbd_bm_resize called with capacity == *
341
kernel: [   *.*] block drbd0: resync bitmap: bits=* words=* pages=*
342
kernel: [  *.*] block drbd0: * *:*:*:* bits:* flags:*
343
kernel: [  *.*] IPVS: Connection hash table configured (size=*, memory=00Kbytes)
344
kernel: [  *.*] nf_conntrack version *.*.* (* buckets, * max)
345
kernel: [  *.*] 0000q: adding VLAN * to HW filter on device evip_macvlan0
346
kernel: [  *.*] IPVS: Registered protocols (TCP, UDP, SCTP, AH, ESP)
347
mmas_syslog_control_setup.sh: Removing MMAS syslog configuration ...
```

Figure 6.3: Template Examples .

In the template examples shown in Figure 6.3, one line is represented by an integer indicating the ID of this template, another line is the full representation of this template. The wildcard symbol * shows that either it is a numeric token or there are more than one unique token at this place across the whole cluster's logs. For example, in the template 325 shown in Figure 6.3, first two wildcards of `[ *.*]` are due to the numeric values representing the time duration from starting, the following three wildcards represent the real time clock, and the last two wildcards represent the date. In the template 328, `*:*:*.*` represents PCI address here. The last wildcard symbol in template 338 shows that there are different tokens, which are `eth0` and `bond0` given in Figure 6.4.

```
kernel: [   *.*] IPv0: ADDRCONF(NETDEV_UP): eth0: link is not ready
kernel: [   *.*] IPv0: ADDRCONF(NETDEV_UP): bond0: link is not ready
```

Figure 6.4: Logs in Cluster 338.

For avoiding leaking Ericsson's data, it is better to not show more log examples here nor in *Chapter 2. Background* as long as they are sufficient enough to illustrate my idea.

6.2 Machine Learning

In the following part of this section, we use *Normal* to represent the log data set with no errors or failures. The term *Failure* refers to the logs from failed test cases.

In this section, the log data used is from the *Blue Night Train* test services, as mentioned in Section 2.1, running over 60 days. Every day, one package of logs collected from the running tested is stored on the server. It is found out that, among these 60 log packages, almost one third of them are containing errors or even did not finish the execution due to some early failure termination, which are the *Failures* mentioned at the beginning of this section. In the rest roughly 40 *Normal* packages, the most recent 13 packages, compared to previous ones, are having big difference in the platform settings and test configurations so that they do not remain the same comparability. Therefore, the exact log data utilized for training and evaluating machine learning models are the system logs from the remaining 27 log packages. Among these 27 packages, 25 of them are selected as training set, one of them is validation set, the last one is chosen as evaluation set. This normal log set for evaluation is referred as *Normal 1.* For the rest 26 normal logs utilized in fold validation, they are named as *Normal 2* to *Normal 27.*

The evaluation data is fixed but the validation set is not. For the 26 data sets for both training and validation, the *fold validation* is also applied in order to avoid the overfitting on validation set. For example, within the 26 data sets, every one of them is labeled by number from 1 to 26. In the first time of train-and-validation process, 1st log data set is chosen for validation and the rest are used for training.

However, in the second time of train-and-validation process, 2ed log data set is chosen for validation and the rest are used for training. Technically speaking, each of the 26 data sets will be the validation set once and consequently there will be 26 train-and-validation processes. This is called 26-fold validation.

Besides this, one additional log package, which previously is removed out due to its erroneousness and failure, is also utilized here for the evaluation step. The intention is to find out the difference between the failed ones and correct ones and, furthermore, to identify the places of errors within the failed test cases. This failed log set for evaluation is referred to *Failure 1*.

For each system log set, there are more than $45,000$ lines of logs. In total, 26 log sets contain more than 1.1 million lines of logs. This provides a good estimation of the problem scale and model size that will be built.

All the machine learning experiments are running on Amazon Web Services (AWS) with GPU support base 64-bit Linux system. The GPU instance - `g2.2xlarge` model is utilized here. It contains 8 high frequency Intel Xeon E5-2670 (Sandy Bridge) processors and one high-performance NVIDIA GPU with $1,536$ CUDA cores and 4GB of video memory. Detailed information regarding the AWS G2 instances can be found at: `http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using_cluster_computing.html`.

6.2.1 RNN Training and Validation

The experiments are conducted based on several modified variables. By monitoring the relevant learning process, the influence of these impactors can be observed and analyzed in order to obtain a better fitting model.

First of all, the sentence length of 20, 40, 60, and 80 are examined. The sentence length, as discussed in Section 5.3, is an important factor having impact on how much time steps the RNN can be back-propagated. It is as same as the term window size mentioned in Section 5.2 for NB. The longer the sentence length is, the more long-term dependency the RNN can learn. However, the training time could be exponentially increasing.

In this experiment, a RNN model with two LSTM layers is built. Each of the layer is having 512 LSTM units. The batch size is 128, which is the number of data samples for each training and gradient descent. The base learning rate is set to 0.001, according to the previous experiments and the recommendation of Keras's official documents. The dropout rate is 0.2. A dropout rate of 0.2 will weaken the effect of overfitting but only to some extent. The impact of dropout will be further examined in the following experiments. Too high dropout rate could undermine the learning capability of RNN as well.

For a certain number of original logs (number of IDs), with different values of step, the number of data samples will also differ. The number of sample is calculated based on Equation 6.1.

$$\#\text{sample} = \frac{\#\text{ID}}{\text{step}} \tag{6.1}$$

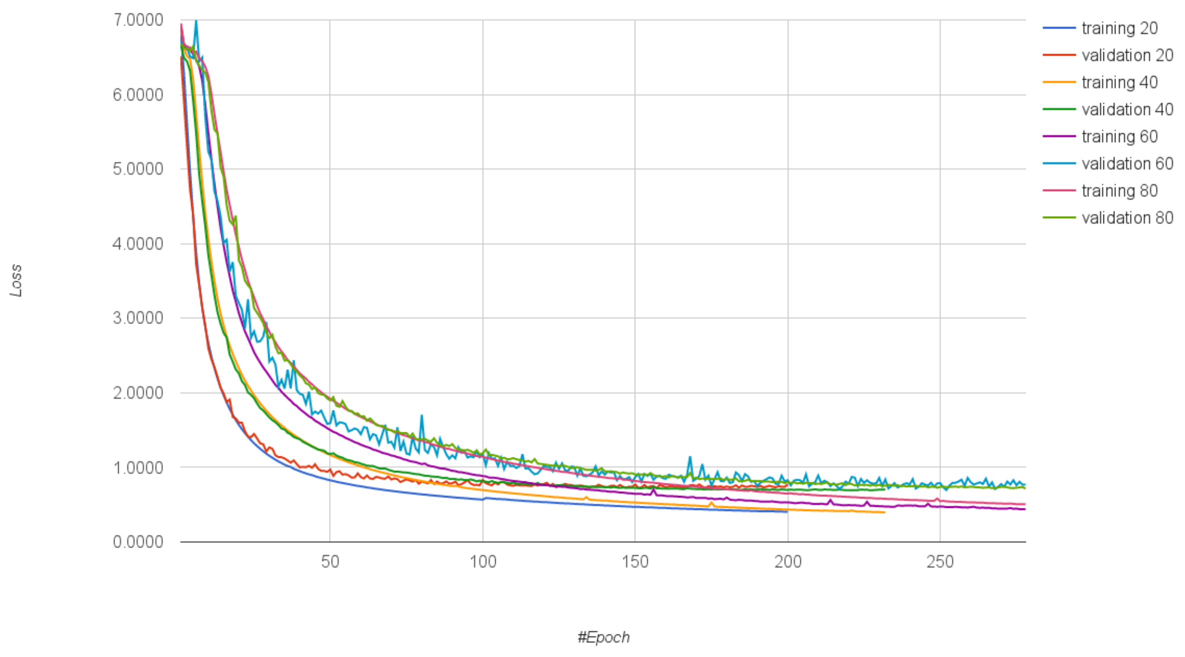


Figure 6.5: RNN Training and Validation - Sentence Length, Loss Curve.

Normally, the step is set as equal to the sentence length. For the set with value less than the sentence length, the learning process of near data samples could be overlapping and slowing down the speed. On the other hands, if the sample step is larger than the sentence length, there must be certain patterns between steps that cannot be learned. For 1.12 million IDs, given the sentence length is 20, the number of samples will be $56,000$. If the sentence length is 80, there will be $14,000$ samples. Setting the batch size as 128 will make sure that there will be always enough number of batch trainings for every epoch (iteration) so that the final loss accuracy value will not be unstable when all values are averaged.

The loss curves are shown in Figure 6.5 and the accuracy curves are shown in Figure 6.6. In these figures, the notion of "`training 20`" means it is the relevant training curve based on the sentence length of 20. Similarly, the notion of "`validation 40`" refers to the validation curve based on the sentence length of 40. For convenience, we refer the term "`experiment-sentence-x`" as the experiment with sentence length of `x`.

It is clear that, the trainings of `experiment-sentence-40` and `experiment-sentence-20` are converging faster than others with longer sample sentence. They are having

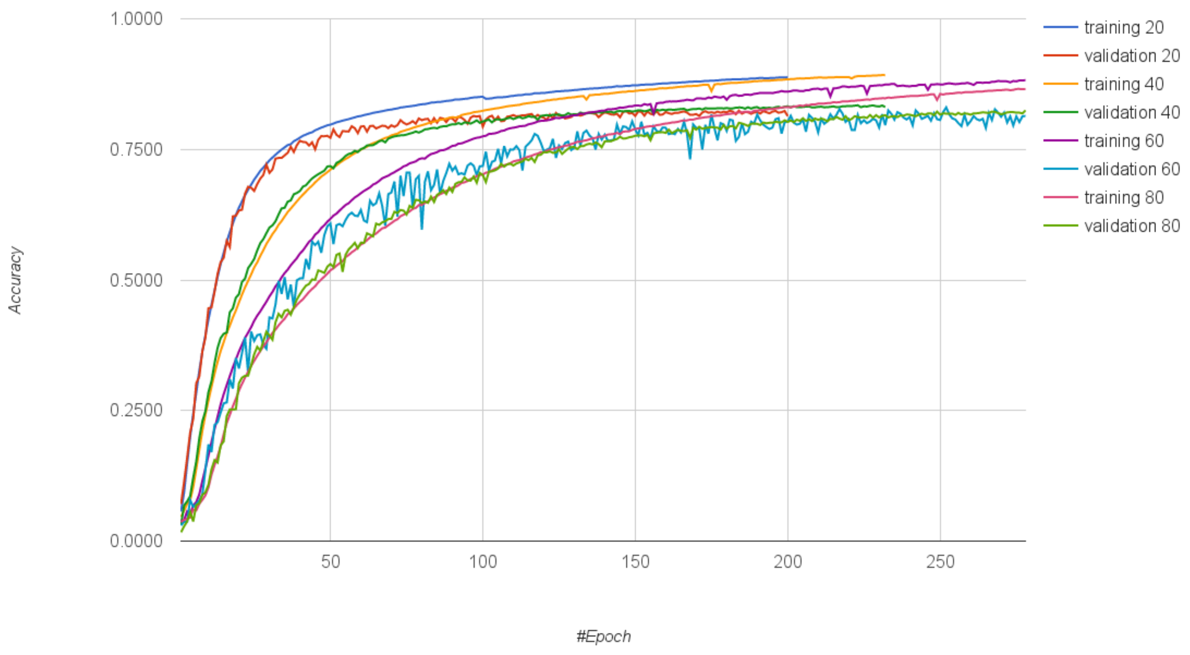
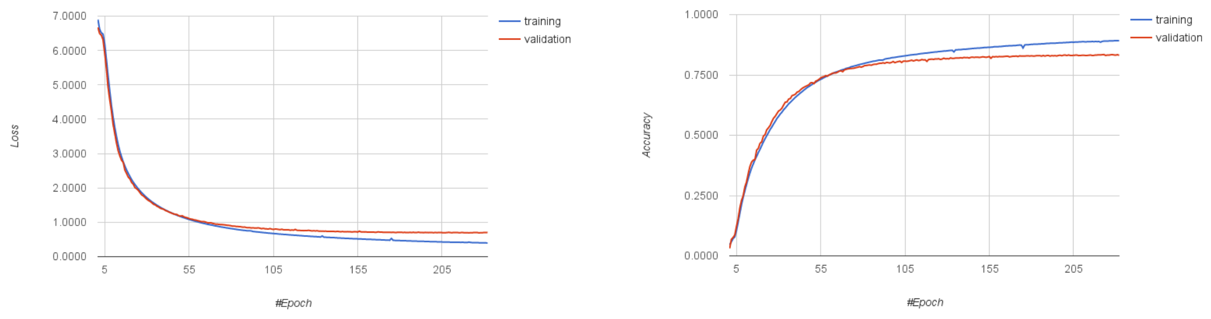


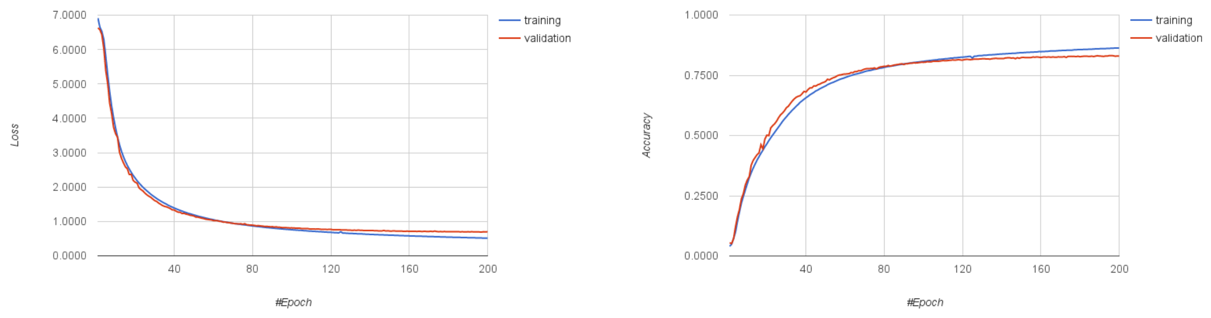
Figure 6.6: RNN Training and Validation - Sentence Length, Accuracy Curve.

less number of epochs and each epoch is also faster. Furthermore, compared with `experiment-sentence-40`, `experiment-sentence-20` shows a worse case with regard to the overfitting since the gap between its training curve and validation curve is larger and happens very early. As indicated in Figures 6.5 and 6.6, with same amount of time, the experiments with smaller sentence length can achieve quite promising results loss and accuracy. However, given sufficient enough amount of training time and iterations, experiments with larger sentence length could lead to a better convergence value; whereas, this is the decision we have to make: whether take incredibly much more time for obtaining a very small optimization. The trade-off made here is to choose sentence length of 40 in the following experiments.

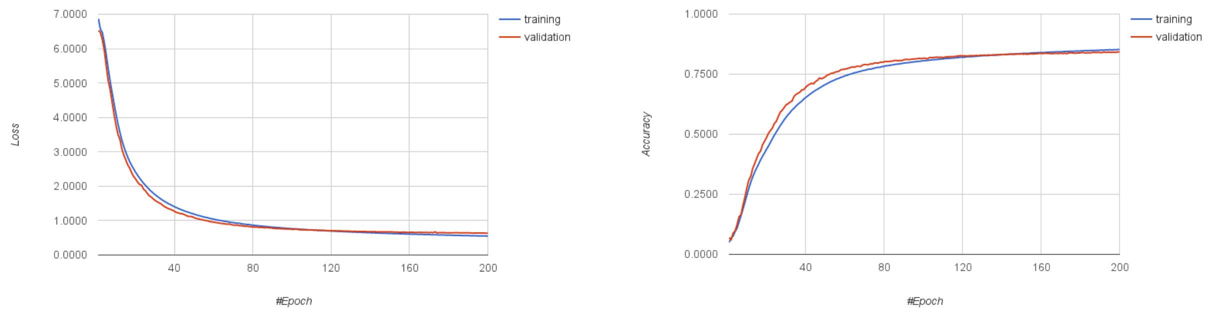
From the Figures 6.5 and 6.6, it is obvious that there is overfitting. The results of `experiment-sentence-40` is individually shown in Figures 6.7a, where the dropout is set to 0.2. Now, the dropout is increased to 0.4 and 0.5, and the loss and accuracy results are shown in Figures 6.7b and 6.7c, respectively. When the dropout is set to 0.5, the overfitting effect is not showing, at least within 200 number of iterations, unlike before. Even though it is uncertain that what will happen after 250 or 300 iterations, there might be still overfitting after a long time training but it is not



(a) Dropout = 0.2.



(b) Dropout = 0.4.



(c) Dropout = 0.5.

Figure 6.7: RNN Training and Validation - Dropout.

concerned in this case because: (i) it is not really necessary to train the model for such as long time just for getting a very little more accuracy, even though the curves have been converging for a while; (ii) it is always inevitable to have overfitting if a RNN model is heavily trained based on certain sets of data and validated on another data set.

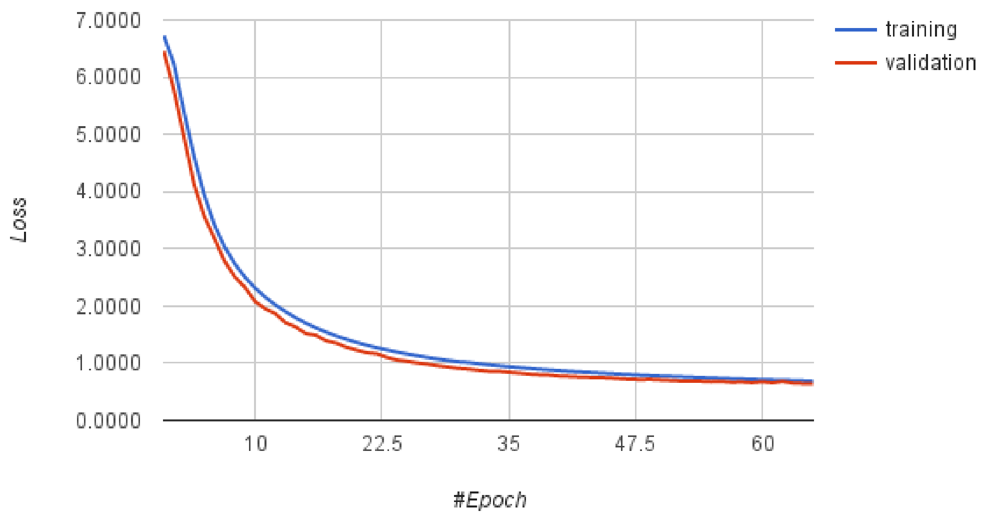


Figure 6.8: RNN Training and Validation - Step=20.

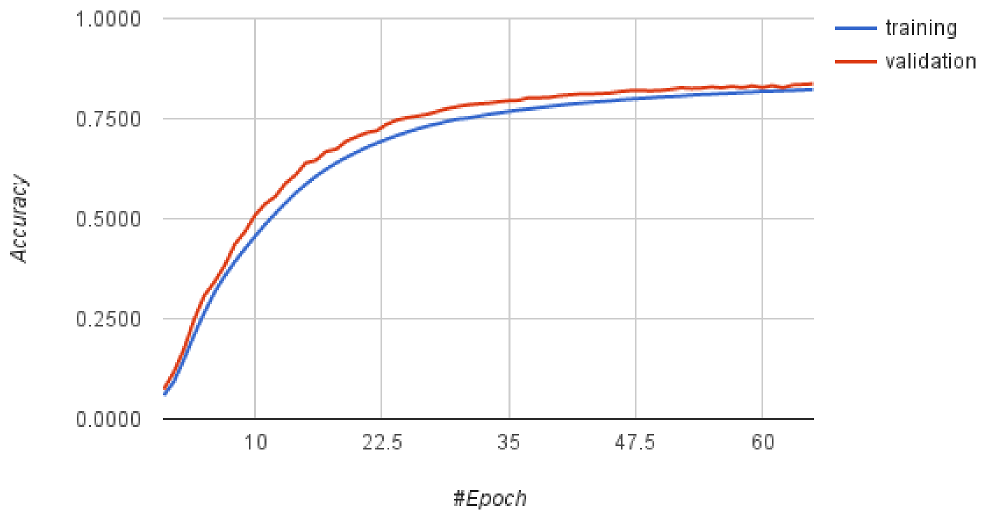


Figure 6.9: RNN Training and Validation - Accuracy Curve, Step=20.

In Figures 6.8 and 6.9, the sampling step is set to 20 instead of being equal to the sentence length, which is 40. Additionally, the dropout rate is 0.5. The overfitting effect is not showing but it is also interesting to see that the number of iterations needed to reach the convergence is around $60 \sim 65$. This is because that the number of data samples for each training and each epoch is doubled and the model is further tuned during each epoch. However, even though the number of iterations is decreased, the time of each iteration is increased accordingly, which makes the total training time barely changed.

6.2.2 RNN vs. Naive Bayes

Since the RNN training and evaluation is incredibly time consuming, it is unrealistic to complete all the 26-fold trainings and validations. Only three fold are conducted and, based on the results, we say that further running is not needed and the conclusions could be offered.

The many-to-many approach demonstrated in Figure 5.11 is applied in this section for the reason that it is more capable of learning consecutive sequence instead of only learning the next symbol, which is illustrated in Section 5.3.2. This choice is also based on some of my initial test, which shows that many-to-many approach normally have $2 \sim 3\%$ more accuracy than many-to-one approach during prediction.

For saving the time consumed on model training, the sequence is not sampled with step of each ID. The *sample step* is defined as the distance between each data samples on the original sequence. If the sample step is equal to d, the Figure 5.11 can be represented as a new Figure 6.10. The $n * d + k + 1$ has to be less or equal

```
        X_train                                  y_train
ID 1,     ID 2,      ..., ID k           ID 2,      ID 3,     ...,  ID k+1
ID d+1,   ID d+2,    ..., ID d+k         ID d+2,    ID d+3,   ...,  ID d+k+1
ID 2d+1, ID 2d+2,    ..., ID 2d+k   ->   ID 2d+2,  ID 2d+3,   ...,  ID 2d+k+1
...
ID n*d+1, ID n*d+2, ..., ID n*d+k        ID n*d+2,  ID n*d+3, ..., ID n*d+k+1
```

Figure 6.10: Many to Many with Sample Step d.

than the length of the original sequence. The hopping sampling reduces the training time of RNN model since the number of samples is d times decreased.

Because of the hopping sampling with step of d, another variable called *offset* is further defined as the very first sample location. If the offset is equal to o, where the offset should be less than the sample step $o < d$, each index of the IDs in Figure 6.10 should be added by o. In this case, the $n * d + k + 1 + o$ has to be less or equal than the length of the original sequence. By having the offset selected randomly during sampling, the whole original sequence can be covered with sufficient times of training.

Nevertheless, the hopping sampling is only applied on RNN model. The NB model is training on every sample step by step since the training and evaluation time of NB is extremely lower than the time of RNN. Therefore, there is no need to apply hopping sampling on NB.

To be more specific regarding the interpretation of accuracy, in this section, it is not considered only based on a single prediction (the most probable output) but also on the top-2 or top-3 options. For example, if the model makes the predictions that the most probable `ID A` is with probability of 41% and the second one `ID B` is with 40%, the model will make the final decision that the correct prediction is `ID A`. However, it is really unfair to say that it is `ID A` if `ID B` is only slightly unlikely compared to `ID A`. Therefore, a quota is defined so that a number of most probable candidates are taken into account. As long as the ground truth is among the several predicted candidates, it is reasonable to say that the model is able to make the correct prediction. Normally, the number of predicted options is one, two or three. Accepting several alternatives as correct predictions is because eventually these predictions will be used to conduct anomaly detection, and only a very unlikely message is relevant to detect.

In this experiment, a RNN model with three LSTM layers is built. Each of the layer is having 512 LSTM units. The batch size is 128, which is the number of data samples for each time of training and gradient descent. The length of sentence is 40, which is the length of each input data sample. The total number of training samples is more than $28,000$ (i.e., more than $28,000 * 40 = 1,120,000$ lines of logs). The number of validation samples is 1170 (i.e., roughly 46,800 lines of logs, which is equal to one single log data set). The step = 40 is equal to the sentence length. If the sample step is larger than the sentence length, there must be certain patterns between steps that cannot be learned. The base learning rate is set to 0.001, according to the previous experiments and the recommendation of Keras's official documents. The dropout rate is 0.5 according to previous experiments.

An overall conclusion can be derived from the results of two Tables 6.3 and 6.4: the RNN is more capable than NB in terms of predicting the next ID given its previous ID subsequence. RNN is obviously outperforming NB in every case: one, two or

	Accuracy	
#candidates	RNN	NB
1	88.2130%	76.1221%
2	93.7655%	86.1736%
3	95.7624%	89.9336%

Table 6.3: RNN vs. NB on Normal Log 2.

three candidates of predictions. The more candidates are taken into account, the smaller the gap is between RNN and NB. For instance, in Table 6.3, the accuracy of RNN is 12.0909%, 7.5919% and 5.8288% better than NB for one, two and three

	Accuracy	
#candidates	RNN	NB
1	89.0287%	76.3373%
2	94.2405%	86.4293%
3	96.0785%	90.1067%

Table 6.4: RNN vs. NB on Normal Log 3.

predicted candidates, respectively.

For NB, based on a same group of training and validation data sets, a logarithms version of it is also run and tested. It is proved that the obtained results (accuracies for one, two and three candidates) are exactly same.

The Figures 6.11 and 6.12 show the learning curves (loss and accuracy) for RNN model. The loss is converging to the level of 0.5 and the accuracy is tending to 0.86%. Additionally, there is hardly sign of overfitting.

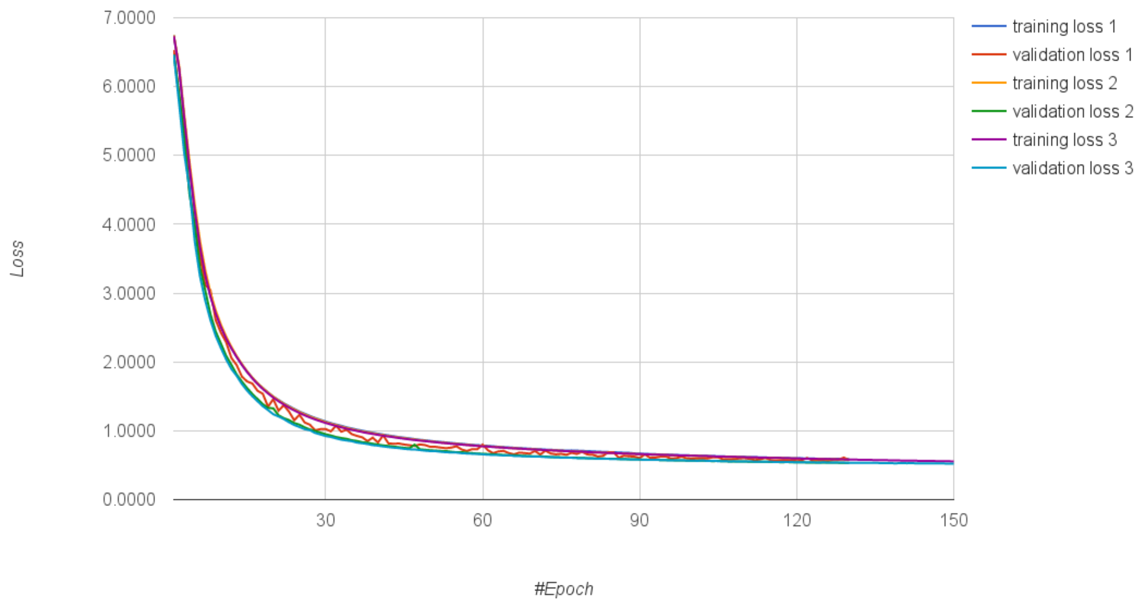


Figure 6.11: RNN Training and Fold Validations - Loss Curve, 3-Fold.

However, the training and evaluation time for RNN and NB are completely in different level, which makes the problem a bit more tricky. Even though RNN is much better than NB in terms of the performance, which is unsurprising, it takes

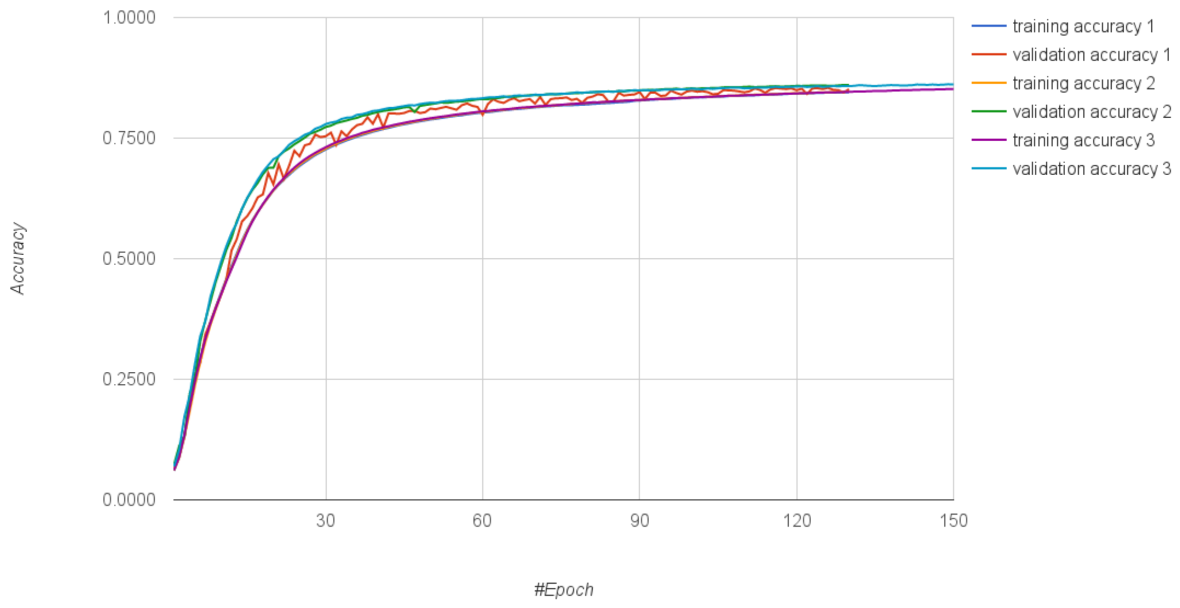


Figure 6.12: RNN Training and Fold Validations - Accuracy Curve, 3-Fold.

more time to train as well. With regard of 25 training log data set, it normally takes at least 7-8 hours to train RNN in order to obtain the convergence of loss and stable accuracy; whereas, NB only needs several minutes to complete the training process for the same amount of time. As to the evaluation time, both RNN and NB could take 1.5 hour for evaluating one single log data set.

6.2.3 Anomaly Detection

From Table 6.5, it is found out that only based on the final results of accuracy, RNN is quite similar for each of the three times of fold validation and for each of the log data set within every fold validation. Therefore, it is quite unlikely to obtain some interesting points for conducting anomaly detection only based on these results.

In order to pin-point the place where errors exactly happened, the probabilities of each of the ID prediction is obtained and plotted on the basis of timeline. In the following part of this section, the plotted figure analysis will be focused on the fold-1. They results of other folds are actually similar.

The probabilities are converted to negative log scale so that the less likely the prediction is, the more outstanding it will be plotted on the graph. The reasoning behind this is that, in this case, it is quite logical to regard the prediction of 0.1

#candidates	Accuracy		
fold-1	Normal 2	Normal 1	Failure 1
1	88.2130%	88.1612%	89.8608%
2	93.7655%	93.8259%	94.9804%
3	95.7624%	95.9041%	96.6546%
fold-2	Normal 3	Normal 1	Failure 1
1	89.0287%	88.8409%	90.3006%
2	94.2405%	94.2252%	95.1624%
3	96.0785%	96.1070%	96.7820%
fold-3	Normal 4	Normal 1	Failure 1
1	88.9451%	89.0459%	90.5237%
2	94.3576%	94.3784%	95.6495%
3	96.2300%	96.2321%	96.3316%

Table 6.5: RNN on several logs.

and 0.0001 probability is with more fundamental difference than there is between probability 0.2 and 0.1. In this project, the number of unique classes to be classified is around 3000 to 4000. Even the random selection could lead to the probability of $1/4000 \sim 1/3000$, which is $0.00025 \sim 0.000333$. This value will be more than 8 after negative logarithmic transformation: 8.294 (0.00025) and 8.007 (0.000333). In the extreme case, the actual probability of 1 is with value of 0 in negative logarithmic scale.

The negative log probability distribution of fold-1 on time for Normal 1 are shown in Figures 6.13. The horizontal axis represents each of the ID on the sequence, which is normally containing more $45,000$ IDs. The vertical axis represents the negative log value of the probability of each next ID prediction.

As discussed in the previous sections, the predictions are conducted by having different number of candidates: one, two and three. Therefore, the figures are also plotted based on different number of candidates. The top left one plots all probabilities for each of the IDs. In the top right figure, the first prediction candidate is set to zero. In the bottom left and the bottom right figures, the top-2 and top-3 candidates are set to zeros, respectively. The reason for doing this is to remove the several the most probable ones and make the anomalies more obvious out of others since the purpose of this experiment is to point out the logs representing these anomalies.

In the last graph of Figures 6.13, the areas where lots of points gathered together would indicate the places where most of the anomalies happen. It is because that the logs (IDs) near each other are having very low probabilities. And many nearby logs with low probability are more likely to show out the system unexpected behaviors than only a few.

However, in these figures, many such places happen but they are not the anomalies if we look back to review the original logs. These experiments are based on the

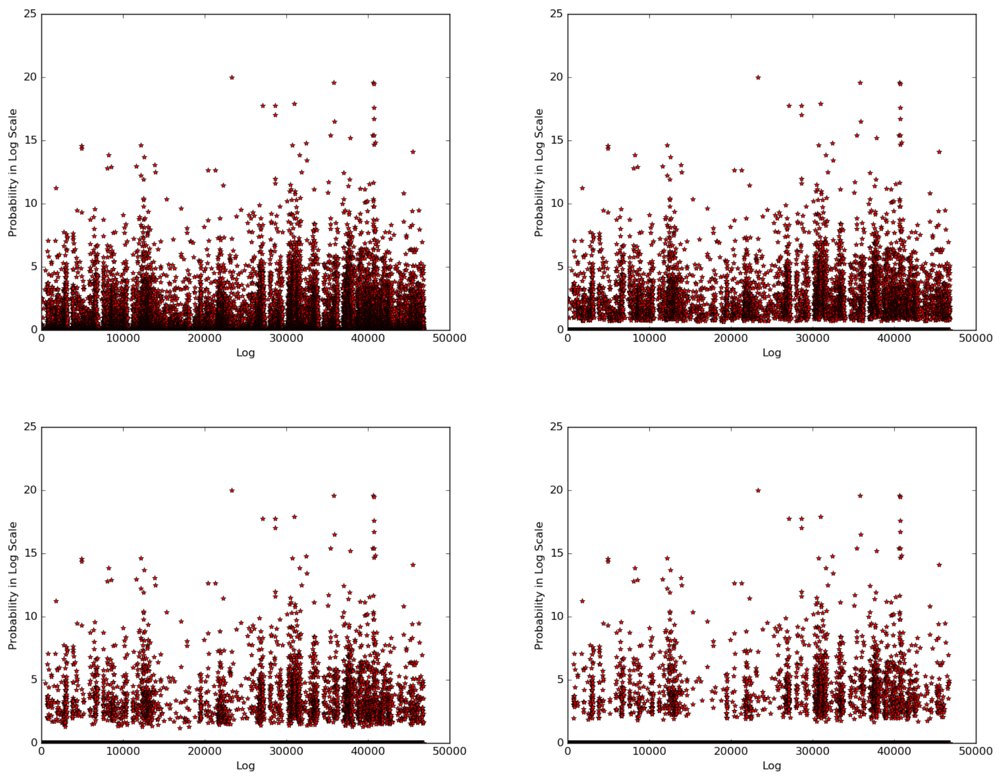


Figure 6.13: Negative Log Probability of Each Next ID Prediction on Time - RNN, Fold-1, Normal 1.

normal log sets and no actual failures exist. Therefore, the expected results are: (i) in Figures 6.13 and 6.14, there should be rarely low probability points gathering together since they are normals; whereas, (ii) in Figures 6.15, there should be one or two certain places indicating the test terminations. After digging deeper into the original logs and consulting the testers and developers at Ericsson, it is found out that these unexpected places are happening during the time when several software components are running in parallel and logging their behaviors in an interleaved way. That means the logs generated from each of the different components are randomly intervened into with each other, which makes the RNN learning much harder and more complex. Theoretically, RNN is capable of handling such issues but with sufficient enough amount of data. However, the combinations of intervened logs are too much to learn and the amount of data we have is also far from sufficient, where there are only less than 30 days of logs. Moreover, if it is essentially random that which message goes before which (i.e., logs from parallel components are essentially randomly intervened with each other), no method at all could deterministically predict the next symbol. It is only possible to tell which symbols are likely to come soon around some time. Then, this is another area. This is actually the extra

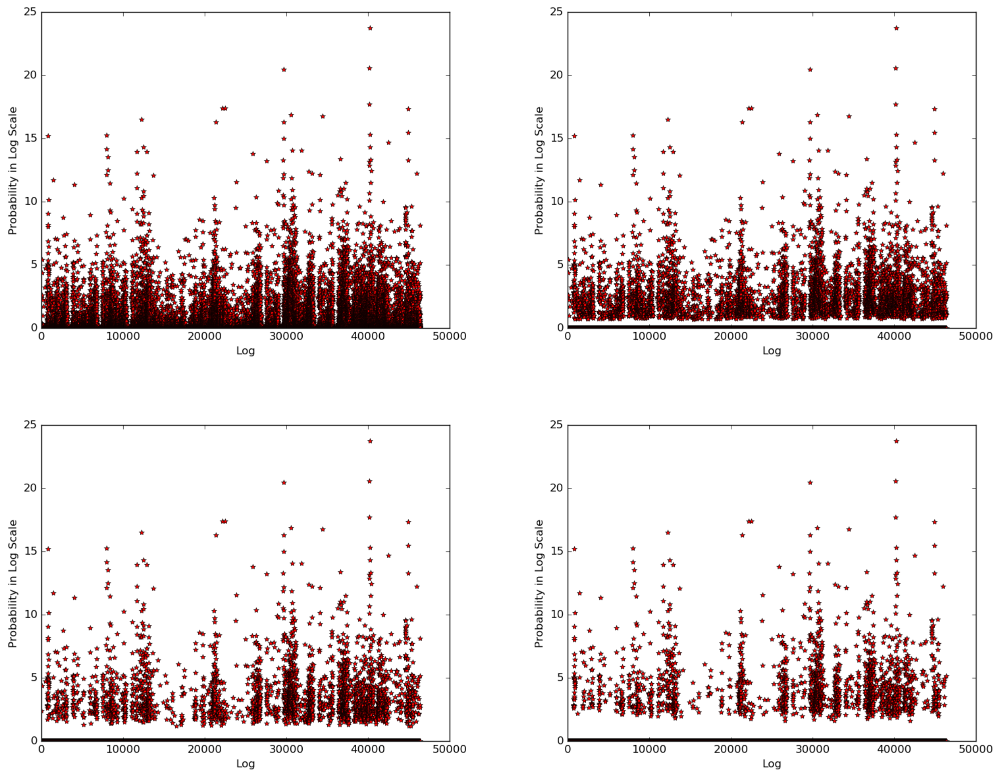


Figure 6.14: Negative Log Probability of Each Next ID Prediction on Time - RNN, Fold-1, Normal 2.

reason for accepting the three first alternatives, but it only works for up to three intertwined log streams.

A simple example of such problem can be demonstrated as in Figure 6.16. As shown in the demonstration, just given three components with several logs for each, the possible combinations could be hundreds, let along with dozens of components each of which is with hundreds or even thousands of logs. 30 days of samples are definitely not enough.

The same negative log probability distribution of fold-1 on time for other log sets can be found in Figures 6.14 for Normal 2 and in Figures 6.15 for Failure 1. The results in Figures 6.14 are very similar to the results in Figures 6.13. The Figures 6.15 shows that the log sequence is much less than other figures since this is based on log *Failure 1*, which terminated at certain time point. It has the same problem discussed before for normal logs: the cluster of low probability logs does not really indicate the failure or anomalies. Additionally, the Figures 6.17 for NB model prediction look even worse and more mass than the results of RNN.

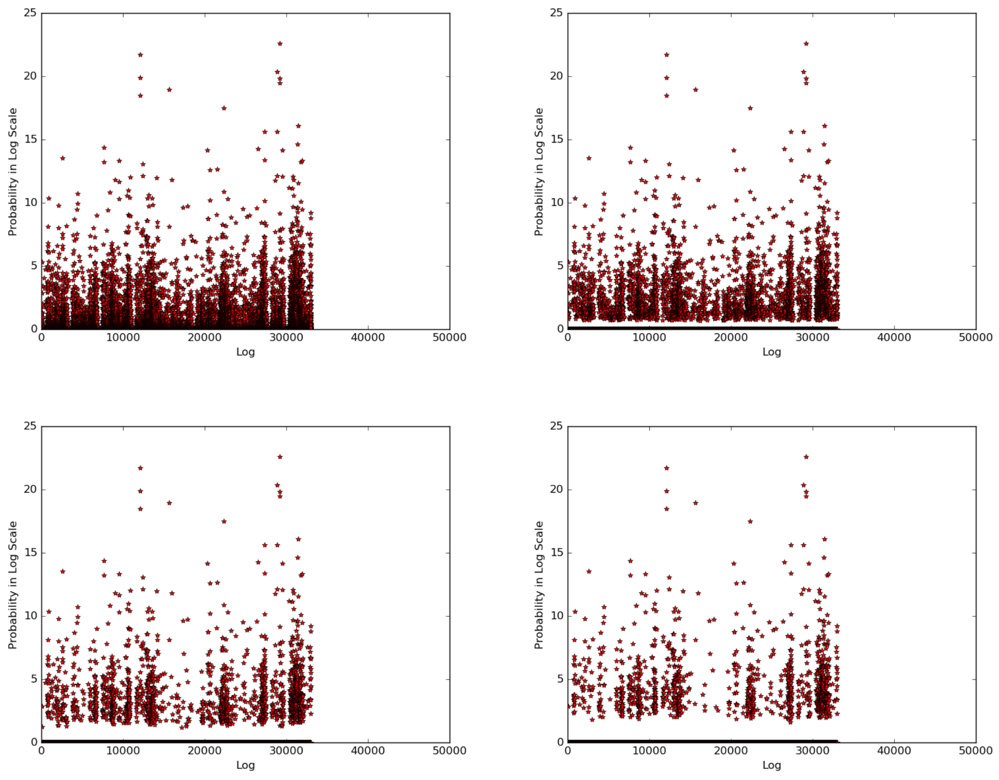


Figure 6.15: Logarithmic Probability of Each Next ID Prediction on Time - RNN, Fold-1, Failure 1.

```
Component A: A1, A2, A3, ...
Component B: B1, B2, B3, ...
Component C: C1, C2, C3, ...

Logs: A1, A2, A3, B1, B2, B3, C1, C2, C3, ...
Logs: A1, B1, C1, A2, B2, C2, A3, B3, C3, ...
...
```

Figure 6.16: Demonstration of the intervened logs.

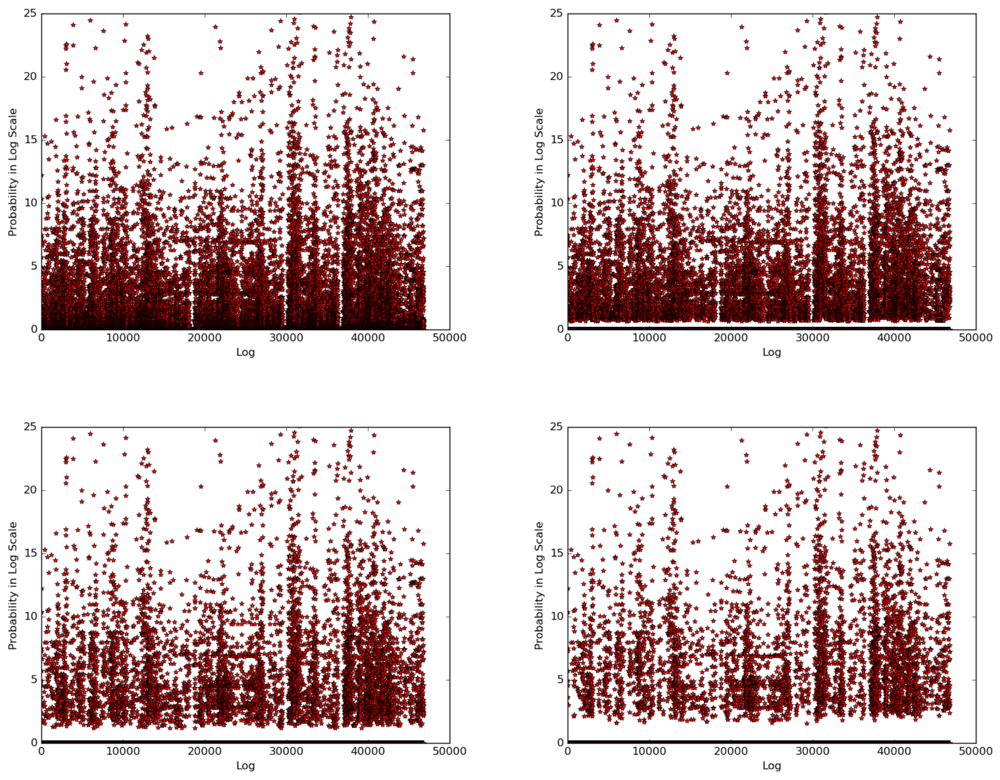


Figure 6.17: Negative Log Probability of Each Next ID Prediction on Time - NB, Normal 2.

Chapter 7

Conclusions and Further Discussion

The objective of this thesis project is to explore a way to conduct log analysis efficiently and effectively by applying relevant data analytics techniques and machine learning algorithms in order to help people quickly detect the places of system failures and anomalies causing the failures. The major works are divided and finished within the following two stages.

In the first stage, a Log Template Extractor is designed and implemented by preprocessing and clustering original logs in order to obtain useful data which can be fed to machine learning algorithms in the second stage. This extractor collects raw log data, clusters them based on their similarity, extracts the log template of each of the clusters, labels the grouped clusters by unique identifications, and finally feeds the labels as input to the next stage. Having the result discussion given in Section 6.1, the achieved templates are satisfying and fulfilling the original expectations. The value of threshold can also be adjusted in order to obtain various levels of log representation abstractions and template extractions. Based on clustered logs, the extractor is able to classify newly imported logs and deal with unknown logs.

In the second stage, RNN models are constructed, trained and experimented in order to learn the sequential patterns from the label sequence obtained from previous stage. Meanwhile, a Naive Bayes (NB) model is built as a baseline for the comparison study. A comparable log analysis, based on these two machine learning algorithms, is conducted for detecting the places of system failures and anomalies causing the failures. An overall conclusion, based on the comparable experiment, is: the RNN is more capable than NB in terms of predicting the next ID given its previous ID subsequence. RNN is outperforming NB in every case: one, two or three candidates of predictions. The more candidates are taken into account, the smaller the gap is between RNN and NB.

In order to pin-point the places where errors exactly happened (i.e., anomaly detection), the probabilities of each of the ID prediction is obtained and plotted on the basis of timeline. Even though the accuracy results obtained are acceptable, especially for the case of three prediction candidates, the models built and trained are not sufficient to conduct anomaly detection for now. Some unexpected results are shown in the plotted figures. For example, some figures indicate the unusual behaviors of the system but they are actually not the anomalies but are false positives. On the contrast, at the places where there should be anomalies, they are overwhelmed by the false positives and messed up with each other. This could be caused by insufficient amount of data or the randomly intervened logs from several software components running in parallel. Moreover, anomaly detection intends to identify the system behavior, of which the probability is very rare. However, our models are not able to achieve such high accuracy of prediction. Therefore, even for the achieved accuracy of 96%, there are still 4% incorrect predictions (which are 2,000 incorrectnesses among 50,000 logs). It is really hard to tell the anomalies from such 2,000 incorrectnesses. And it will lead to the disordered figures plotted in Section 6.2.3.

Even though the final results for anomaly detection are not as what have been expected, this thesis project still has its meaningful achievements. The Log Template Extractor is created for log data processing and clustering. Several RNN and NB models are researched and implemented for dealing with log sequence learning and prediction. The comparable study is thus given for evaluating RNN and NB accordingly. During this process, many theories and practices of RNN, NB and other relevant machine learning knowledges are deeply analyzed, particularly based on our study case. Although the results of anomaly detection are not good, it explores a new way of conducting log analysis.

In the future, the possible works could be increasing the amount of log data available, resolving the problem of intervened logs, digging deeper into RNN in order to train and evaluate the models in a better way, or even finding other suitable machine learning algorithms.

Bibliography

[1] Xia Cai, Michael R. Lyu, Kam-Fai Wong, and Roy Ko. Component-based software engineering: technologies, development frameworks, and quality assurance schemes. In *Software Engineering Conference, 2000. APSEC 2000. Proceedings. Seventh Asia-Pacific*, pages 372–379. IEEE, 2000.

[2] Grady Booch. *Object Oriented Design with Applications.* Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1991.

[3] Tomohiro Kimura, Koji Ishibashi, Takayoshi Mori, Hideyuki Sawada, Tsuyoshi Toyono, Ken Nishimatsu, Atsuyori Watanabe, Akihiro Shimoda, and Kohei Shiomoto. Spatio-temporal factorization of log data for understanding network events. In *INFOCOM, 2014 Proceedings IEEE*, pages 610–618. IEEE, 2014.

[4] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710, 1966.

[5] Fred J. Damerau. A technique for computer detection and correction of spelling errors. *Communications of the ACM*, 7(3):171–176, 1964.

[6] Richard W. Hamming. Error detecting and error correcting codes. *Bell System technical journal*, 29(2):147–160, 1950.

[7] Tatsuaki Kimura, Akio Watanabe, Tsuyoshi Toyono, and Keisuke Ishibashi. Proactive failure detection learning generation patterns of large-scale network logs. In *Network and Service Management (CNSM), 2015 11th International Conference on*, pages 8–14. IEEE, 2015.

[8] Risto Vaarandi et al. A data clustering algorithm for mining patterns from event logs. In *Proceedings of the 2003 IEEE Workshop on IP Operations and Management (IPOM)*, pages 119–126, 2003.

[9] Thomas Reidemeister, Miao Jiang, and Paul AS Ward. Mining unstructured log files for recurrent fault diagnosis. In *Integrated Network Management (IM), 2011 IFIP/IEEE International Symposium on*, pages 377–384. IEEE, 2011.

[10] Risto Vaarandi and Kãrlis Podiņš. Network ids alert classification with frequent itemset mining and data clustering. In *Network and Service Management (CNSM), 2010 International Conference on*, pages 451–456. IEEE, 2010.

[11] Leonardo Mariani and Fabrizio Pastore. Automated identification of failure causes in system logs. In *Software Reliability Engineering, 2008. ISSRE 2008. 19th International Symposium on*, pages 117–126. IEEE, 2008.

[12] Jon Stearley. Towards informatic analysis of syslogs. In *Cluster Computing, 2004 IEEE International Conference on*, pages 309–318. IEEE, 2004.

[13] Adetokunbo Makanju, Stephen Brooks, A Nur Zincir-Heywood, and Evangelos E Milios. Logview: Visualizing event log clusters. In *Privacy, Security and Trust, 2008. PST'08. Sixth Annual Conference on*, pages 99–108. IEEE, 2008.

[14] Thomas Reidemeister, Mohammad A Munawar, and Paul AS Ward. Identifying symptoms of recurrent faults in log files of distributed information systems. In *Network Operations and Management Symposium (NOMS), 2010 IEEE*, pages 187–194. IEEE, 2010.

[15] Risto Vaarandi. Mining event logs with slct and loghound. In *Network Operations and Management Symposium, 2008. NOMS 2008. IEEE*, pages 1071–1074. IEEE, 2008.

[16] Risto Vaarandi. A breadth-first algorithm for mining frequent patterns from event logs. In *Intelligence in Communication Systems*, pages 293–308. Springer, 2004.

[17] Risto Vaarandi. *Tools and Techniques for Event Log Analysis.* Tallinn University of Technology Press, 2005.

[18] Rene De La Briandais. File searching using variable length keys. In *Papers presented at the the March 3-5, 1959, western joint computer conference*, pages 295–298. ACM, 1959.

[19] Peter Brass. *Advanced data structures.* Cambridge University Press Cambridge, 2008.

[20] Jon Stearley. Sisyphus log data mining toolkit. `http://www.cs.sandia.gov/sisyphus/`, 2009. [Online; access permission needed].

[21] Adetokunbo AO Makanju, A Nur Zincir-Heywood, and Evangelos E Milios. Clustering event logs using iterative partitioning. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1255–1264. ACM, 2009.

[22] Adetokunbo Makanju, A Nur Zincir-Heywood, and Evangelos E Milios. A lightweight algorithm for message type extraction in system application logs. *Knowledge and Data Engineering, IEEE Transactions on*, 24(11):1921–1936, 2012.

[23] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In *ACM Sigmod Record*, volume 29, pages 1–12. ACM, 2000.

[24] Isidore Rigoutsos and Aris Floratos. Combinatorial pattern discovery in biological sequences: The teiresias algorithm. *Bioinformatics*, 14(1):55–67, 1998.

[25] Aris Floratos and Isidore Rigoutsos. On the time complexity of the teiresias algorithm. *IBM TJ Watson Research Center. Yorktown Heights. New York*, 1998.

[26] Brona Brejová, Chrysanne DiMarco, Tomáš Vinar, Sandra Romero Hidalgo, Gina Holguin, and Cheryl Patten. Finding patterns in biological sequences. *Unpublished project report for CS798G, University of Waterloo, Fall*, 2000.

[27] Andreas Wespi, Marc Dacier, and Hervé Debar. *An intrusion-detection system based on the Teiresias pattern-discovery algorithm.* IBM Thomas J. Watson Research Division, 1999.

[28] Tongqing Qiu, Zihui Ge, Dan Pei, Jia Wang, and Jun Xu. What happened in my network: mining network events from router syslogs. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 472–484. ACM, 2010.

[29] Edward Chuah, Shyh-hao Kuo, Paul Hiew, William-Chandra Tjhi, Gary Lee, John Hammond, Marek T Michalewicz, Terence Hung, and James C Browne. Diagnosing the root-causes of failures from cluster log files. In *High Performance Computing (HiPC), 2010 International Conference on*, pages 1–10. IEEE, 2010.

[30] Edward Chuah, Gary Lee, William-Chandra Tjhi, Shyh-Hao Kuo, Terence Hung, John Hammond, Tommy Minyard, and James C Browne. Establishing hypothesis for recurrent system failures from cluster log files. In *Dependable, Autonomic and Secure Computing (DASC), 2011 IEEE Ninth International Conference on*, pages 15–22. IEEE, 2011.

[31] George H John and Pat Langley. Estimating continuous distributions in bayesian classifiers. In *Proceedings of the Eleventh conference on Uncertainty in artificial intelligence*, pages 338–345. Morgan Kaufmann Publishers Inc., 1995.

[32] Ashraf M Kibriya, Eibe Frank, Bernhard Pfahringer, and Geoffrey Holmes. Multinomial naive bayes for text categorization revisited. In *AI 2004: Advances in Artificial Intelligence*, pages 488–499. Springer, 2004.

[33] Andrew McCallum, Kamal Nigam, et al. A comparison of event models for naive bayes text classification. In *AAAI-98 workshop on learning for text categorization*, volume 752, pages 41–48. Citeseer, 1998.

[34] Kamal Nigam and Rayid Ghani. Analyzing the effectiveness and applicability of co-training. In *Proceedings of the ninth international conference on Information and knowledge management*, pages 86–93. ACM, 2000.

[35] David D Lewis. Naive (bayes) at forty: The independence assumption in information retrieval. In *Machine learning: ECML-98*, pages 4–15. Springer, 1998.

[36] Ted Pedersen. A simple approach to building ensembles of naive bayesian classifiers for word sense disambiguation. In *Proceedings of the 1st North American chapter of the Association for Computational Linguistics conference*, pages 63–69. Association for Computational Linguistics, 2000.

[37] Gerard Escudero, Lluís Màrquez, and German Rigau. Naive bayes and exemplar-based approaches to word sense disambiguation revisited. *arXiv preprint cs/0007011*, 2000.

[38] Harry Zhang. The optimality of naive bayes. *AA*, 1(2):3, 2004.

[39] Rich Caruana and Alexandru Niculescu-Mizil. An empirical comparison of supervised learning algorithms. In *Proceedings of the 23rd international conference on Machine learning*, pages 161–168. ACM, 2006.

[40] Pierre Simon marquis de Laplace. *Théorie analytique des probabilités*. V. Courcier, 1820.

[41] Harold Jeffreys. *The theory of probability*. OUP Oxford, 1998.

[42] Mark Webster. Bayesian statistics. *Journal of Applied Statistics*, 40(12):2773–2774, 2013.

[43] Michael Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

[44] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[45] Frank Rosenblatt. Principles of neurodynamics. perceptrons and the theory of brain mechanisms. Technical report, DTIC Document, 1961.

[46] Deeplearning4j. Introduction to deep neural networks. `http://deeplearning4j.org/neuralnet-overview.html`, 2016. Online; accessed 7 June 2016.

[47] Stanford CS231n. Cs231n: Convolutional neural networks for visual recognition. `http://cs231n.github.io/`, 2016. Online; accessed 7 June 2016.

[48] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 807–814, 2010.

[49] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. Rectifier nonlinearities improve neural network acoustic models. In *Proc. ICML*, volume 30, page 1, 2013.

[50] Ian J Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron Courville, and Yoshua Bengio. Maxout networks. *arXiv preprint arXiv:1302.4389*, 2013.

[51] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep learning. Book in preparation for MIT Press, 2016.

[52] Chun-tian Cheng, Wen-jing Niu, Zhong-kai Feng, Jian-jian Shen, and Kwok-wing Chau. Daily reservoir runoff forecasting method using artificial neural network based on quantum-behaved particle swarm optimization. *Water*, 7(8):4232–4246, 2015.

[53] Peter Sadowski. Notes on backpropagation. `https://www.ics.uci.edu/~pjsadows/notes.pdf`, 2016. Online; accessed 11 June 2016.

[54] Andrej Karpathy. The unreasonable effectiveness of recurrent neural networks. `http://karpathy.github.io/2015/05/21/rnn-effectiveness/`, 2015. Online; accessed 13 June 2016.

[55] Denny Britz. Recurrent neural networks tutorial. `http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/`, 2015. Online; accessed 13 June 2016.

[56] Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[57] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. *arXiv preprint arXiv:1211.5063*, 2012.

[58] incomplete PROVISIONAL and SUPPLEMENT TO SHARE LECTURE. Machine learning and bio-inspired optimization.

[59] Sepp Hochreiter, Yoshua Bengio, Paolo Frasconi, and Jürgen Schmidhuber. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies, 2001.

[60] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *Neural Networks, IEEE Transactions on*, 5(2):157–166, 1994.

[61] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[62] Felix A Gers and Jürgen Schmidhuber. Recurrent nets that time and count. In *Neural Networks, 2000. IJCNN 2000, Proceedings of the IEEE-INNS-ENNS International Joint Conference on*, volume 3, pages 189–194. IEEE, 2000.

[63] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. *Neural computation*, 12(10):2451–2471, 2000.

[64] Christopher Olah. Understanding lstm networks. `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`, 2015. Online; accessed 13 June 2016.

[65] Klaus Greff, Rupesh Kumar Srivastava, Jan Koutník, Bas R Steunebrink, and Jürgen Schmidhuber. Lstm: A search space odyssey. *arXiv preprint arXiv:1503.04069*, 2015.

[66] Zou Hui and Hastie Trevor. Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society*, 67(2):301–320, 2005.

[67] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[68] Hiroyuki Tanaka. editdistance. `https://github.com/aflc/editdistance`, 2013.

[69] Heikki Hyyrö. Explaining and extending the bit-parallel approximate string matching algorithm of myers. Technical report, Citeseer, 2001.

[70] Stanley F Chen and Joshua Goodman. An empirical study of smoothing techniques for language modeling. In *Proceedings of the 34th annual meeting on Association for Computational Linguistics*, pages 310–318. Association for Computational Linguistics, 1996.

[71] François Chollet. Keras. `https://github.com/fchollet/keras`, 2015.

[72] François Chollet. Keras official document. `http://keras.io/`, 2015.

[73] Andrej Karpathy. Multi-layer recurrent neural networks (lstm, gru, rnn) for character-level language models in torch. `https://github.com/karpathy/char-rnn`, 2015. Online; accessed 30 June 2016.