BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

DEEP LEARNING FOR LOG FILE ANOMALY DETECTION USING DEEPLOG

TERM PROJECT

AUTHOR **Bc. JAKUB KUZNÍK**

SUPERVISOR **doc. Ing. PETR MATOUŠEK, Ph.D.,M.A.**

BRNO 2024

Abstract

Log data is an important and valuable resource for understanding system status and performance issues. The various system logs serve as excellent sources of information for online monitoring and anomaly detection. However, these data are typically large, unannotated, and live, making manual processing very challenging. Numerous supervised and unsupervised methods exist for detecting anomalies in log data. Nevertheless, systems that produce log data are often dynamic and generate large amount of logs. Normal behavior can also evolve over time, rendering traditional methods unsuitable for real-life systems as they were usually learned on outdated system logs, which may not reflect the current system behavior. Min Du et al. propose a Deep Learning supervised method called DeepLog, which employs learning and can adapt to live system log traffic. In this work, we utilize DeepLog for log data anomaly detection and discuss its strengths, weaknesses, and potential enhancements.

Keywords

DeepLog, Anomaly detection, Deep Learning, Log Files

Reference

KUZNÍK, Jakub. *Deep Learning for Log File Anomaly detection using DeepLog.* Brno, 2024. Term project. Brno University of Technology, Faculty of Information Technology. Supervisor doc. Ing. Petr Matoušek, Ph.D.,M.A.

Contents

Chapter 1

Introduction to Log Analysis for Anomaly Detection

Modern systems produce a large amount of log data, which is very hard to process. These systems are usually designed to operate on a 24x7 basis. Any incident in these systems, including service outage or degradation of service quality, can lead to significant revenue loss. Anomaly detection aims to find anomalies in the behavior of these systems, which is an important part of incident management [3]. Traditionally, developers manually inspect system logs or create rules based on their domain knowledge to detect anomalies, often using keyword search or regular expression matching. However, the complexity of modern systems, their volume, and large-scale implementation involving many individuals and third-party software make it difficult to identify anomalies effectively [5].

The process of log analysis for anomaly detection comprises four main steps: log collection, log parsing, feature extraction, and anomaly detection.

1.1 Log Collection

Log collection is the initial step in log analysis, where raw log records are gathered from the running system. These records, consisting of plaintext log messages indicating the events that have occurred and timestamps, provide the foundational data for subsequent analysis stages [3].

1.2 Log Parsing

Log parsing involves extracting important information from plaintext logs and transforming these text messages into structured data known as events, each with specific parameters. For an event, we usually need predefined templates, which may look like this: `Received block * of size * from *`.

There are two main types of log parsing methods: clustering-based and heuristic-based. In clustering-based log parsers, distances between logs are calculated first, and clustering techniques are often employed to group logs into different clusters. Finally, an event template is generated from each cluster. On the other hand, heuristic-based approaches count the occurrences of each word on each log position. Next, frequent words are selected and composed as the event candidates, which are then chosen to be the log events.

1.3 Feature Extraction

Feature extraction involves extracting valuable attributes from data, reducing the dimensionality of the data, and encoding events with methods such as One-Hot encoding to create suitable feature vectors for data mining tasks. This process also includes grouping events using methods like fixed windows, which have a fixed size based on the timestamp.

Then there is the sliding window, where the window has a second parameter called the step size. The window moves, which is smaller than the window size, meaning that one event can be in multiple windows.

Lastly, session windows are based on identifiers instead of the timestamp, aiming to capture different execution paths in the system to generate an event count matrix [**?**].

1.4 Anomaly Detection

Lastly, anomaly detection involves feeding the matrix into machine learning models, which can be supervised or unsupervised. For training the supervised models, we first need to annotate the data. After the process of training, the model can be applied to new incoming sequences of logs. Finding the most suitable method is a very difficult task, as they usually focus on the target system [3].

For anomaly detection, we can choose from supervised or unsupervised data mining methods, as well as others like reinforcement learning. These methods can be applied in conjunction with various log parsing and feature extraction approaches. In the context of anomaly detection, the primary distinction between supervised and unsupervised methods lies in the requirement for data annotation during training. Supervised methods needs annotation such as „anomaly“ or „not anomaly“ label for each log or window, which are typically unavailable in real-life systems. Finding a way to annotate the data for training can be challenging due to the large volume of logs, making unsupervised methods more widely applicable. It’s important to note that while unsupervised methods offer greater versatility, they can also pose greater challenges in effective implementation. Examples of supervised learning methods include Logistic Regression, Decision Trees, SVM and DeepLog while unsupervised learning methods encompass techniques such as Log Clustering, PCA, and Invariants Mining [2].

Chapter 2

DeepLog

DeepLog is a deep learning, data-driven framework for anomaly detection that harnesses a large volume of logs. A fundamental concept of DeepLog involves interpreting log entries as elements of a sequence following specific patterns and grammar rules. System logs, indeed, originate from a set of programs adhering to finite states and control flows, bearing resemblance to natural language in several aspects. DeepLog utilizes a Recurrent Neural Network featuring Long Short-Term Memory (LSTM), a component commonly employed in language models. This LSTM architecture enables DeepLog to grasp both long-term and short-term dependencies within the data, thereby autonomously learning log patterns from normal system executions. Following an initial training phase, DeepLog can be deployed for system anomaly detection. A notable advantage of DeepLog lies in its capability for incremental updates, facilitating adaptation to new log patterns and changes over time. This adaptability is achieved through DeepLog's mechanism for user feedback: if a normal log entry is misclassified as an anomaly, users can flag it as a false detection, prompting dynamic updates to the model's weights [1].

2.1 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) possess the capability to transmit information backward from one layer to another, as depicted in Figure 2.1. This allows the network's current input to be influenced by its previous inputs, thereby creating contextual memory. The previous inputs are represented in blue, demonstrating how the rightmost input incorporates information from all preceding inputs. Each input generates its own output and forwards information to the subsequent iteration in the picture. This mechanism finds extensive utility in natural language processing, where the meaning of a word often depends on preceding words in the next layer. During backpropagation, the error is propagated in the opposite direction of all the arrows in Figure 2.1. In the second picture there is a simplified scheme of the RNN. Notably, there are no actual cycles present in this process, even though it may appear so. If we expand the cycle as shown in the first picture, we actually have multiple copies of the same neural network. This highlights the strong relationship between RNNs and data in sequences and lists, such as log data [8].

[1] https://colah.github.io/posts/2015-08-Understanding-LSTMs/[accessed on April 17, 2024]

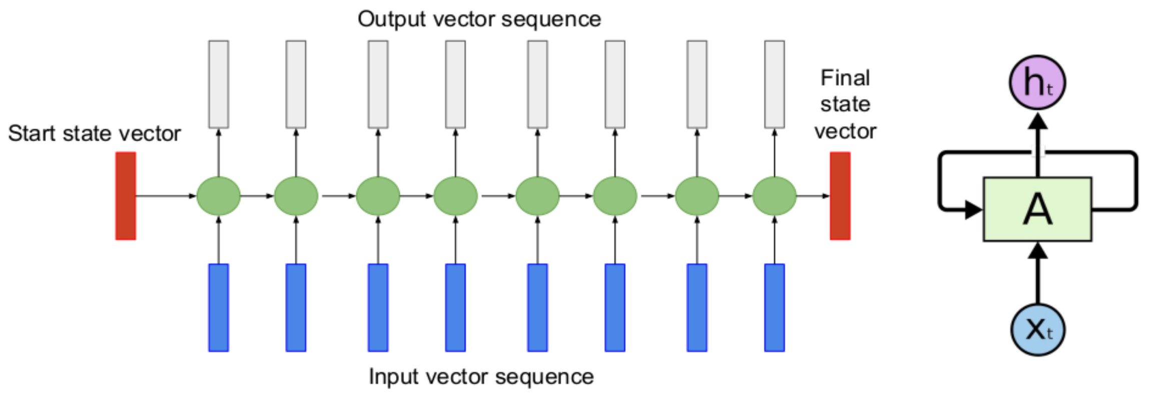

Figure 2.1: Recurent Neural Network architecture[1]

2.2 Long Short-Term Memory

One of the main ideas behind RNNs is that they may be capable of connecting previous information to the present task. However, in reality, they encounter difficulties in remembering information that occurred a long time ago, often retaining only short-term information. For example, when processing language, RNNs can predict the next word in a sentence, such as „My name *,“ but they struggle to predict information mentioned earlier in the text, such as completing the sentence „My name is *.“ This limitation underscores the challenge of long-term dependency modeling in RNNs. In theory, RNNs should be capable of capturing such long-term dependencies, but in practice, they often fail to learn them [7]. For addressing such long-term dependencies, Hochreiter et al. proposed the idea of Long Short-Term Memory (LSTM) [4].

LSTMs are a special kind of RNN, capable of learning long-term dependencies. LSTMs work quite well on a big variety of problems. All RNNs have a form of chain of blocks where each block is basically one (Neural Network) NN, as shown in Figure 2.2 [7]. We can observe that there is no complicated logic; these are simply concatenated NNs.

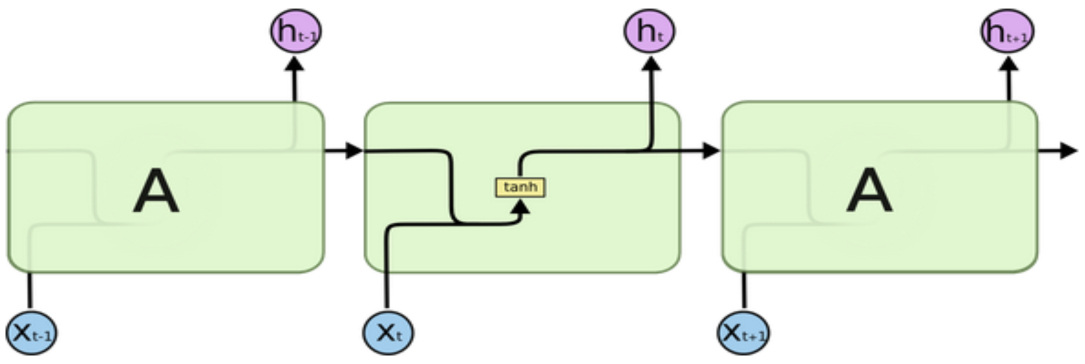

Figure 2.2: Repeating block in a standard RNN and its internals[2]

[2]https://colah.github.io/posts/2015-08-Understanding-LSTMs/ [accessed on April 17, 2024]

On the other hand, we have LSTMs, as shown in Figure 2.3. Instead of a single neural network layer with one activation function, LSTMs consist of four NN layers with four activation functions. The upper line passes the information unchanged, and the lower line serves as a kind of memory, often referred to as the hidden state. LSTMs include four gates in hidden layers responsible for modifying the cell state. These gates enable LSTMs to add optional information to this line. The first sigmoid gate determines how much information (ranging from 0 to 1) should be allowed to pass. Then, there is a second sigmoid gate combined with tanh, which decides and maintains some long-term state. Finally, there is one more sigmoid gate combined with tanh, determining which information should proceed to the next state. In summary, this mechanism provides the block with a kind of cache that can be used for storing long-term information [7].

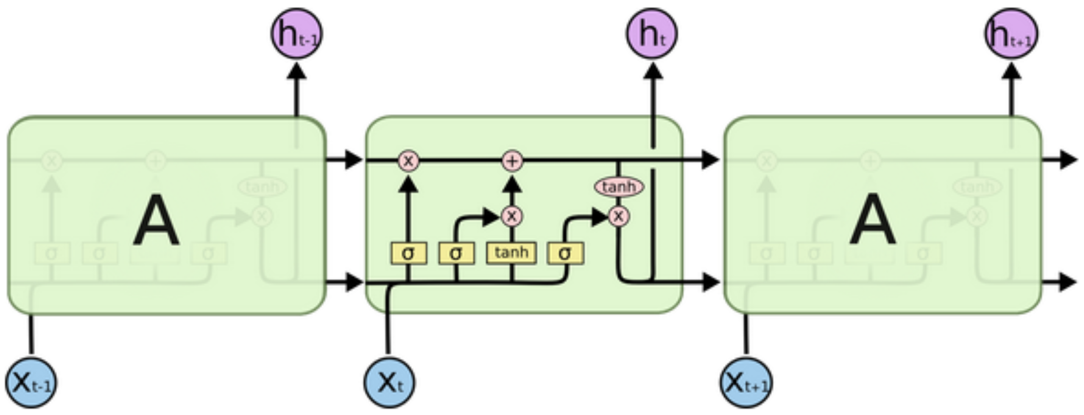

Figure 2.3: Repeating block in a LSTM and its internals[3]

2.3 DeepLog Architecture

DeepLog is an data-driven framework using supervised deep learning, designed to detect anomalies by processing extensive log data. At its core, DeepLog operates on the principle of treating log entries as sequential elements governed by distinct patterns and grammatical structures. Deeplog uses both log keys, and log values with timestamp [1].

The architecture of DeepLog, illustrated in Figure 2.4, comprises three main components: the log key anomaly detection model, the parameter value anomaly detection model, and the workflow model for anomaly detection. During the training stage, log entries from normal system execution paths serve as the training data for DeepLog. Each log entry undergoes parsing into a log key and a parameter value vector. The log key sequences parsed from training logs are utilized by DeepLog to train a log key anomaly detection model and to construct system execution workflow models for diagnostic purposes. Additionally, for each distinct key, DeepLog trains and maintains a model to detect system performance anomalies reflected by metric values. These models are trained using the parameter value vector sequence corresponding to each key [1].

The system first employs the log key anomaly detection model to verify if the incoming log key is normal. If it is, DeepLog proceeds to examine the parameter value vector using

[3]`https://colah.github.io/posts/2015-08-Understanding-LSTMs/` [accessed on April 17, 2024]

the parameter value anomaly detection model specific to that log key. If either the log key or the parameter value vector is predicted as abnormal, the entry is labeled as an anomaly. In such cases, DeepLog's workflow model provides users with semantic information to diagnose the anomaly. As execution patterns may evolve over time or were not present in the original training data, DeepLog offers the option for collecting user feedback [1].

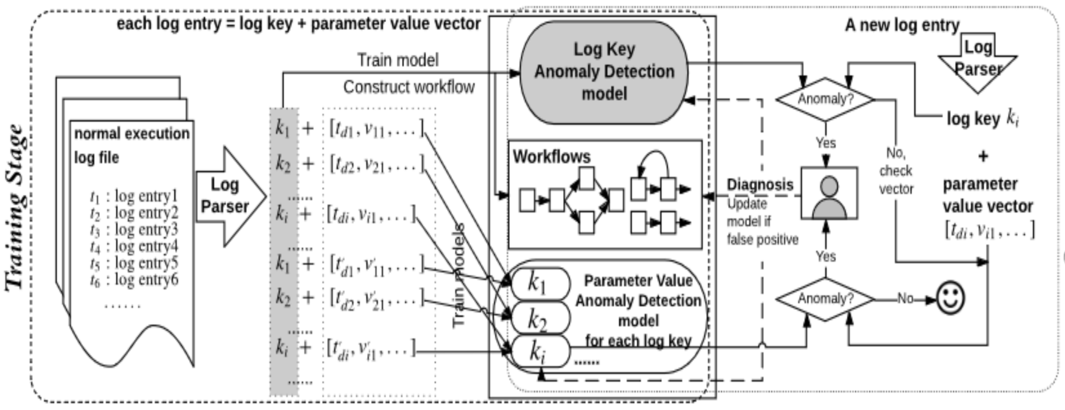
each log entry = log key + parameter value vector
Training Stage
normal execution log file
t_1 : log entry1
t_2 : log entry2
t_3 : log entry3
t_4 : log entry4
t_5 : log entry5
t_6 : log entry6
Log Parser
k_1 + $[t_{d1}, v_{11}, \dots]$
k_2 + $[t_{d2}, v_{21}, \dots]$
k_i + $[t_{di}, v_{i1}, \dots]$
k_1 + $[t'_{d1}, v'_{11}, \dots]$
k_2 + $[t'_{d2}, v'_{21}, \dots]$
k_i + $[t'_{di}, v'_{i1}, \dots]$
Train model
Construct workflow
Train models
Log Key Anomaly Detection model
Workflows
Parameter Value Anomaly Detection model for each log key
k_1
k_2
k_i
Diagnosis
Update model if false positive
Anomaly?
Yes
No, check vector
No
A new log entry
Log Parser
log key k_i
+
parameter value vector
$[t_{di}, v_{i1}, \dots]$
Detection Stage

Figure 1: DeepLog architecture.

Figure 2.4: DeepLog architecture [1]

The log key anomaly detection is implemented using LSTM blocks, and the output of this is a probability distribution of having log key value $k_i \in K$ as the next log key value. Given a sequence of log keys, a LSTM network is trained to maximize the probability of having $k_i \in K$ as the next log key value, as reflected by the training data sequence. Once training is completed, we can forecast the output for an input $w = \{m_{t-h}, ..., m_{t-1}\}$ using a layer comprising h LSTM blocks. Each log key in w feeds into a corresponding LSTM block within this layer. By stacking multiple layers and utilizing the hidden state of the preceding layer as the input for each corresponding LSTM block in the subsequent layer, it transforms into a deep LSTM neural network.

Then there is parameter value anomaly detection for identifying irregular parameters in log entries. In DeepLog, all the log key parameters are placed into a matrix, where each vector represents a separate time series. DeepLog utilizes another LSTM neural network for this purpose. By providing these parameters to DeepLog, it can differentiate between different processes and executions in the system.

Both NN are using the sliding window for the log keys and values. The objective function for training both these networks tries to adjust the weights to minimize the error between a prediction and an observed parameter value vector using Mean Square Error (MSE) and gradient descent.

Finally, there is the online updating of DeepLog, where users can update the weights of DeepLog by evaluating the anomalies reported by the model.

DeepLog has this mechanism for providing feedback to the model, which is great for live systems and it can differentiate between log sequences from various sources. However, it requires a lot of time to prepare and label data for training, as well as a significant amount of computational power to train the method, as neural networks typically demand. Also, we have parameters like LSTM layers, LSTM hidden features, window size, neural network weights, learning rate, which can be quite hard to balance [1].

Chapter 3

Our Implementation of DeepLog

We propose am implementation of DeepLog, where the log key anomaly detection model and the parameter value anomaly detection model are merged into one deep LSTM neural network. This integration aims to add simplicity and consolidate all related information into a single computing model, which often works very well for deep neural networks. In theory, a deep enough network should learn all normal log processes. Additionally, we implement our own log parsing mechanism using existing templates, which can be easily extended. The implementation itself was done using the PyTorch library[1]. All the data are encoded using one-hot encoding and then processed using a sliding window with a step size of 1. This means that one log can be in multiple input windows, adding a form of data augmentation.

3.1 Architecture

In Figure 3.1, we can see our architecture where there is an input matrix of size $m \times z$, followed by n LSTM blocks and one fully connected linear layer with a sigmoid activation function which gives us probability of an anomaly.

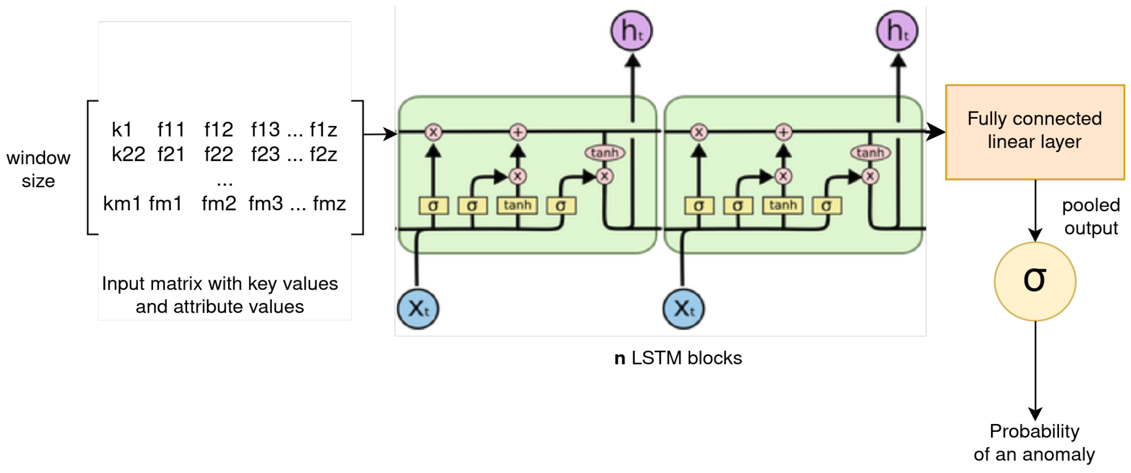

Figure 3.1: Architecture of the model.

[1] https://pytorch.org/ [accessed on April 17, 2024]

The input matrix is a matrix with m log keys and z attribute values for each log key, where the number of rows m represents the window size. Attributes are encoded in the form of one-hot encoding. This matrix is then passed to the first of n LSTM layers. We are using a sliding window with a step size of 1, which means that if we have a window size of 3 with keys $(k1, k2, k3)$, then the next window will be $(k2, k3, k4)$.

Then there are n LSTM blocks for capturing the long-term and short-term relationships between log entries. These blocks have an input size of $z+1$, where 1 stands for the log key and z represents the attribute values. After the LSTM blocks have captured the features of the data, there is one fully connected layer that adjusts the dimension of the input data to the appropriate dimension for the sigmoid function. Lastly, the sigmoid function will give us the probability of the input being an anomaly or normal.

3.2 Model Parameters and Learning Strategy

There are many configurable parameters to consider while implementing DeepLog: window size, optimization function, learning rate, number of LSTM layers, epochs, and imbalance constant. Finding the proper ones, ideally computationally easier, can be very challenging. Due to the lack of computational resources, we only used 2 LSTM layers, a window size of 10, and approximately 5000 epochs. It should be noted that better results can be achieved by adding more layers, epochs, or by simply using more data for training.

The imbalance constant addresses the problem of the small amount of anomalies in the training data. We experimentally chose the value 5, which means that if the anomaly exists in the data and the network fails to detect it, the penalty will be much higher.

Since this problem is just a classification task, we don't use the MSE loss function; instead, we used Cross-entropy with the Adam optimization algorithm with learning rate of 1×10^{-3}.

When setting parameters, we must determine our goal: whether to prioritize maximum accuracy or emphasize anomaly detection while tolerating false positives. This can be achieved by configuring the optimization function, learning rate, and imbalance constant.

3.3 Dataset

For the evaluation, we utilized the Hadoop Distributed File System (HDFS) dataset[2], designed to run on commodity hardware [9, 11]. This dataset has been widely studied, facilitating easy comparison of our results with different studies. Additionally, the dataset includes predefined templates, such as:

```
E1 <>:<> Served block blk_<> to <>
```

These templates were used for parsing the log entries into the log keys and values, where $E1$ represents the log key and $*$ represents the values. The more values and keys we have, the larger the matrix will become, as everything is basically categorical attribute and we need to use one-hot encoding.

[2]https://github.com/logpai/loghub [accessed on April 17, 2024]

Chapter 4

Evaluation

The evaluation has been difficult since processing the whole dataset requires a large amount of computational resources. Therefore, we used a representative subset of the original data for training and a different subset for testing. Then, we compared our results with the baseline DeepLog, PCA [10], and Invariant Mining [6], that were tested on the same dataset. For the evaluation, we used standard statistical metrics: Precision, Recall, and F-measure [1]. In the following equations, TP represents True Positive, FN represents False Negative, and FP represents False Positive.

$$Precision = \frac{TP}{TP+FP} \quad Recall = \frac{TP}{TP+FN}$$

$$F\text{-}measure = \frac{2 \times Precision \times Recall}{Precision + Recall}$$

The results of our experiments, compared to the original work and different methods, are shown in Figure 4.1. DeepLog, PCA, and IM were tested on the whole dataset (11 million log entries), with DeepLog being trained on about 110,000 log entries. In contrast, our method was tested on just 18,000 entries and trained on about 700 due to a lack of computational resources, making it difficult to identify all the normal patterns of the system.

For the experiment, we used a sliding window size of $w = 10$, epochs $e = 5000$, 2 LSTM layers ($l = 2$), an imbalance factor of $i = 5$, the Cross-entropy loss function, and the Adam optimization algorithm with a learning rate of 1×10^{-3}.

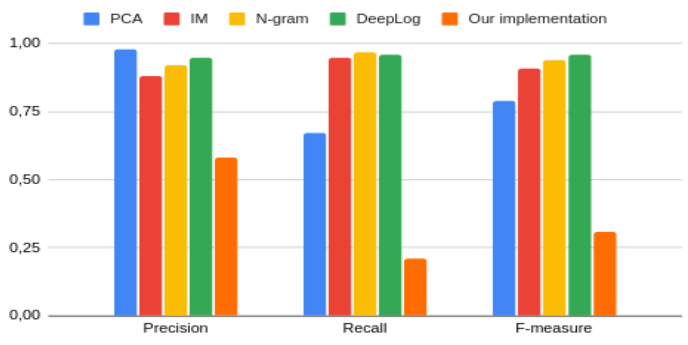

Figure 4.1: Accuracy on HDFS

The exact results of the experiment is shown in the table below 4.1.

Statistic	**Value**
True Negatives (TN)	9808
False Negatives (FN)	5546
True Positives (TP)	1550
False Positives (FP)	1080

Table 4.1: Test Data Statistics

Although our results were not as promising as those of other methods, some trends are evident, and we can certainly achieve much better results with a larger amount of training.

Chapter 5

Conclusion

DeepLog is indeed a very promising method, with accuracy surpassing that of other methods. Its adaptability to changes in system behavior makes it highly suitable for real-life usage. However, the challenge with this method lies in setting proper parameters and obtaining the necessary training data. Nonetheless, DeepLog requires much less data for training compared to other supervised methods. Our modification of DeepLog appears promising, particularly with the potential inclusion of additional LSTM layers and an increase in the volume of training data. Despite our implementation not yielding initially promising results, it has introduced new ideas. With greater computational power, better results are certainly achievable.

Bibliography

[1] Du, M., Li, F., Zheng, G. and Srikumar, V. DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning. In:. October 2017, p. 1285–1298. DOI: 10.1145/3133956.3134015. ISBN 978-1-4503-4946-8.

[2] Farzad, A. and Gulliver, T. A. Unsupervised log message anomaly detection. *ICT Express*. september 2020, vol. 6, p. 229–237. DOI: 10.1016/j.icte.2020.06.003.

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

[4] Hochreiter, S. and Schmidhuber, J. Long Short-term Memory. *Neural computation*. december 1997, vol. 9, p. 1735–80. DOI: 10.1162/neco.1997.9.8.1735.

[5] Lin, Q., Zhang, H., Lou, J.-G., Zhang, Y. and Chen, X. Log clustering based problem identification for online service systems. In:. May 2016, p. 102–111. DOI: 10.1145/2889160.2889232.

[6] Lou, J.-G., Fu, Q., Yang, S., Xu, Y. and Li, J. Mining invariants from console logs for system problem detection. january 2010.

[7] Olah, C. *Understanding LSTM Networks* [https://colah.github.io/posts/2015-08-Understanding-LSTMs/]. 2015. Accessed: April 17, 2024.

[8] Schmidt, R. M. *Recurrent Neural Networks (RNNs): A gentle Introduction and Overview*. 2019.

[9] Xu, W., Huang, L., Fox, A., Patterson, D. and Jordan, M. Detecting Large-Scale System Problems by Mining Console Logs. In: *Proc. of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*. 2009.

[10] Xu, W., Huang, L., Fox, A., Patterson, D. and Jordan, M. Online System Problem Detection by Mining Patterns of Console Logs. In:. December 2009, p. 588–597. DOI: 10.1109/ICDM.2009.19.

[11] Zhu, J., He, S., He, P., Liu, J. and Lyu, M. R. Loghub: A Large Collection of System Log Datasets for AI-driven Log Analytics. In: *IEEE International Symposium on Software Reliability Engineering (ISSRE)*. 2023.