An LSTM Recurrent Neural Network Approach to Anomaly Detection in Hadoop Log Files

Sophia Bulcock
Master of Data Science
University of British Columbia Okanagan
Kelowna, British Columbia, Canada
sophia.bulcock@gmail.com

Graham Kerford
Master of Data Science
University of British Columbia Okanagan
Kelowna, British Columbia, Canada
graham.kerford@gmail.com

ABSTRACT

Servers like Apache Hadoop generate an enormous number of logs which are far beyond the processing speed of human beings. The paper introduces the supervised approach of a Long Short-Term Memory recurrent neural network to identify patterns in log files, and correctly classify anomalies. We design the LSTM RNN after trying unsuccessful statistical based approaches, like Principal Component Analysis, Random Forest and MCLUST clustering. The LSTM is a specific RNN architecture whose design makes it easier to train by its ability to learning long-term dependencies. In this work, we aim to determine whether the LSTM architecture is optimal, and if it is able to correctly classify anomalous and non-anomalous log file entries based on their block ID's. We found that the LSTM recurrent neural network performed better than the statistical based approaches and was optimized by increasing the embedded and hidden dimensions and using a learning rate between 1e-3 and 1e-4. This resulted in a training and testing accuracy of 99.13% and 99.03% respectively.

INTRODUCTION

Log files are used to provide a history of the actions performed by a system and the results of those actions. In this paper, we will be explicitly talking about Apache Hadoop log files, which are generated by Hadoop's jobtracker, namenode, secondary namenode, datanode, and tasktracker [1], which is publicly available. Since the running state of a system is recorded in a log file over time, log file data is naturally time series data, therefore the log file used in this project is naturally sorted ascending by date and time [2]. Log data is a valuable resource for anomaly detection, which can help identify faults within servers. Given the fact that systems, like Apache Hadoop, are on 24/7, this equates to a massive amount of log data being generated, which can quickly become difficult to manage. The log file used in this paper contains 11 million log lines generated from Apache Hadoop servers, therefore manual parsing and anomaly detection would not be possible. We hope to use supervised machine learning to build a model that can correctly identify anomalies within this log file, and others like it. Anomalies within log files will typically produce a pattern that records malicious behaviors, which should differ from the system's normal log patterns [2]. Traditionally, detecting anomalies would rely on an administrator to manually analyze the log text, but with millions of lines, this becomes unfeasible [2]. This is why we propose to use an LSTM recurrent neural network to parse these logfile lines and detect anomalies based on pattern recognition. Machine learning to detect log file anomalies can use supervised and unsupervised methods, and since we have the labeled data on hand, we chose to use a supervised technique. In summary, we propose to assign numeric character values to each type of message contained within a logline, group these messages by block ID, assign whether an anomaly was present or not, and use this to train supervised LSTM RNN to detect anomalies in these log files.

LITERATURE REVIEW AND BACKGROUND

Deep machine learning and statistical approaches are promising techniques used to identify anomalies in log file data. Statistical based approaches that have been suggested for the classification of these anomalies are Principal Component Analysis (PCA) and Factor Analysis (FA). Other techniques that have been proposed in the past are Random Forest, Local Outlier Factor, and K-Means clustering [3]. We began with trying Principal Component Analysis on a subset of the data, followed by Random Forest, based on the literature. Next, we decided to try PCA & Mclustering, followed by a basic neural net in R. The results were not promising, so we decided to move on to an LSTM recurrent neural network.

Our proposed method to detect anomalies in this log file data is to sort the message types occurring in the log files into numeric characters, then group this data into block IDs, which produces a string vector of the sequence of numbers/messages that occur with a block ID. We then propose to split the data into training, testing, and validation sets and train an LSTM recurrent neural network with this training data. This technique was chosen because the data, which is now in character strings, is an ideal candidate for natural language processing, which LSTM has been widely used for [4]. Additionally, these types of neural networks are popular in log file anomaly detection and have been widely used. An experiment done by Zhao *et al.* [2], which involved testing the effectiveness of LSTM's in log file anomaly detection through contrast experiments with principal component analysis, support vector machines, and gaussian mixture models, found that the LSTM neural network outperformed all the other techniques. This makes sense because log file anomalies should be identifiable by patterns of words that do not occur as often as normal word patterns. This evidence made it an easy choice to make LSTM our supervised learning method of choice for Hadoop log file anomaly detection.

Long Short-Term Memory, abbreviated LSTM, is an artificial recurrent neural network architecture used in the field of deep learning. This type of recurrent neural network was developed by Hochreiter & Schmidhuber in 1997 [5]. Unlike traditional neural networks, which are not recurrent, RNN's have loops in them that allow for information to persist and be remembered over time [6]. What makes an LSTM recurrent neural network different from other types of recurrent neural networks, is its ability to learning long-term dependencies [5]. Recurrent neural networks, abbreviated RNN, have a type of chain-like structure that makes them very similar to sequences and lists, which makes them a good candidate for the architecture of choice when dealing with this type of data [6]. However, some of the drawbacks of standard recurrent neural networks are that they suffer from both exploding and vanishing gradients due to their iterative nature [6]. A standard RNN's gradient is equal to its recurrent weight matrix raised to a higher power [6]. Since these matrix powers are iterated over during training, they cause the gradient to grow and shrink at a rate that is exponential in the number of time steps [6]. The issue with exploding gradients is that they can lead to unstable networks, and even in extreme cases NaN values. This is the first drawback of using a standard recurrent neural network, and one of the reasons we chose to use an LSTM recurrent neural network instead.

The second, and much larger issue with standard RNN's is their vanishing gradient problem. What this essentially means is that the components in the gradient that have to do with long-term dependencies/memory are small, and the components in the gradient that have to do with short-term dependencies/memory are large [6]. Simply put, the RNN's learning with short-term dependencies is optimized, but this is not the case with the long-term dependencies [6]. When we are dealing with log data that is millions of lines long, and we want to optimize the long-term dependencies to make better predictions with the model. It has been measured that standard RNN's are said to fail to learn in the presence of past observations that are greater than 5 to 10 time steps [7]. These drawbacks to standard RNN's are why we chose to use an LSTM recurrent neural network, which avoids this vanishing gradient problem by re-parametrizing the RNN [6].

The architecture of an LSTM RNN is unique as compared to other RNN's and simple neural networks. The first unique feature is its memory cells. LSTM RNN's have memory cells for modeling the long-range dependencies, which in turn avoids the vanishing gradient problem that standard RNN's suffer from [4]. In more basic terms, this means the LSTM RNN's are built for remembering information for long periods. Since our log file has eleven million lines, ideally it needs to be able to remember and recognize patterns for a longer period, as in many log lines, to make accurate predictions. This memory cell feature is what made an LSTM recurrent neural network our RNN of choice to detect anomalies in Hadoop log files.

An LSTM recurrent neural net is built to solve sequential data problems, due to its specific memory cells, which can store information that can be recurrently connected [8]. The basic architecture of an LSTM recurrent neural consists of self-connected cells which each have input gates, forget gates, and output gates [8]. An LSTM block, therefore, consists of cells, and one block can have one or many cells. However, a cell belongs to only one LSTM block. The input, forget and output gates are a way to optionally let information through and they are composed of a sigmoid neural net layer and a pointwise multiplication operation (Hadamard product multiplication) [5].

The input to the LSTM cell input at time t is x_t and the corresponding values for the input gate are denoted by the equation below [8]. This layer decides which values will be updated.

$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i)$$

The corresponding equation for the forget gate and the output gate is denoted by the two equations below [8].

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f)$$

$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o)$$

In these equations, W and U represent the weight matrices, b represents the bias vector and σ represents the sigmoid activation function used in the LSTM [8]. The forget gate layer to the LSTM analyzes the h_{t-1} and x_t, and outputs a number between 0 and 1 for each number in the cell state [5]. A 0 represents a number that will be completely thrown away, and 1 represents a number that will be completely kept [5]. The block input at time t is represented by the tanh layer equation written below [8].

$$\hat{C}_t = \tanh\,(W_C x_t + U_C h_{t-1} + b_c)$$

In the tanh layer equation above, W_C and U_C represent the weight matrices, b_c represents the bias vector and tanh represents the hyperbolic tangent activation function [8]. This creates a vector of new candidate values $\hat{C}_t$, that could potentially be added to the cells state. The state of the LSTM cell at time t is then represented by the equation below, with the $\circ$ symbol representing the Hadamard product multiplication of the matrices [8]. This is the new cell state.

$$C_t = f_t \circ C_{t-1} + i_t \circ \hat{C}_t$$

Lastly, the LSTM RNN block's output at time t is denoted by the following equation [8]. The tanh forces the values to be between -1 and 1, and this is multiplied by the output of the cell state above, to generate the block's output, h_t.

$$h_t = o_t \circ \tanh(C_t)$$

METHODOLOGY

The data used for this analysis was uploaded via a log file as a single string. The string was subdivided into 11 million individual logs by row, figure 1. Each log can be future subdivided into 6 columns, 'Date', 'Time', 'PID', 'Level', 'Component', and 'Content', table 1.

081109 204106 329 INFO dfs.DataNode$PacketResponder: PacketResponder 2 forblock blk_-6670958622368987959 terminating

Figure 1. Single log extracted by row from the log file as a single string.

Table 1. Columns produced from the logs.

Date	Time	PID	Level	Component	Content
081109	204106	329	INFO	dfs.DataNode$PacketResponder	PacketResponder 2 for block blk_-6670958622368987959 terminating

Numerical values were assigned to the Level, Component, and Content of each log as the 'Level_code', 'Component_code', and 'Tag_code', respectively. These codes act as indicators for each category of each column. The Level_code, Component_code and TypeOfContentCode, tables 2, 3, & 4, respectively, were based on individual categories within each individual column. The Tag_code however is based on the combination of the content and component columns, summarized in the 'Tag' column, table 4. The initial dataframe is summarized in Table 6.

Table 2. unique levels and their corresponding level_code.

Level	Level_code
'Info'	0
'Warn'	1

Table 3. unique Components and their corresponding Component_code.

Component	Component_code
'dfs.DataBlockScanner'	0
'dfs.DataNode'	1
'dfs.DataNode$DataXceiver'	2
'dfs.DataNode$PacketResponder'	3
'dfs.FSDataset'	4
'dfs.FSNamesystem'	5

Table 4. unique TypeOfContent and their corresponding TypeOfContent_code.

TypeOfContent	TypeOfContentCode
'BLOCK* NameSystem.addStoredBlock:'	0
'BLOCK* NameSystem.allocateBlock:'	1
'BLOCK* ask'	2
'Deleting block'	3
'PacketResponder 0'	4
'PacketResponder 1'	5
'PacketResponder 2'	6
'Received block'	7
'Receiving block'	8
'Served'	9
'Starting thread'	10
'Verification succeeded'	11

Table 6. unique Tags and their corresponding Tag_code.

Tag	Tag_code
'BLOCK* NameSystem.addStoredBlock:'	0
'BLOCK* NameSystem.allocateBlock:'	1
'BLOCK* ask'	2
'Deleting block'	3
'PacketResponder 0'	4
'PacketResponder 1'	5
'PacketResponder 2'	6
'Received block'	7
'Receiving block'	8
'Served'	9
'Starting thread'	10
'Verification succeeded'	11

Table 7. Summary of the dataframe following the initial wrangling of the data.

	Date	Time	PID	Level	Component	Content	TypeOfContent	datetime_id	Tag	Level_code	Component_code	TypeOfContent_code	Tag_code
0	2008-11-09	203615	148	INFO	dfs.DataNode$PacketResponder	PacketResponder 1 for block blk_38865049064139...	PacketResponder 1	81109203615	PacketResponder 1	0.0	3.0	7.0	0
1	2008-11-09	203807	222	INFO	dfs.DataNode$PacketResponder	PacketResponder 0 for block blk_-6952295868487...	PacketResponder 0	81109203807	PacketResponder 0	0.0	3.0	6.0	1
2	2008-11-09	204005	35	INFO	dfs.FSNamesystem	BLOCK* NameSystem.addStoredBlock: blockMap upd...	BLOCK* NameSystem.addStoredBlock:	81109204005	BLOCK* NameSystem.addStoredBlock:	0.0	5.0	0.0	2

Subsequently, the block IDs for each log were extracted from the content column used to group the dataset. Each of the codes were concatenated together based on their block ID, all other columns were removed from the dataframe. A label was added to the dataframe to distinguish between anomaly and normal, table 8. The labels were imported via a csv and were merged to the dataframe based on their block id. This dataframe was used for the initial models used to determine which type of machine learning should be used for the analysis.

Table 8. Dataframe with the merged column codes

	blkID	Tag_code	Component_code	Level_code	Label
0	blk_-1030832046197982436	2	5	0	Normal
1	blk_-1046472716157313227	4	4	0	Normal
2	blk_-1049340855430710153	8 7	3	0	Normal

In preparation to input, the data into the LSTM, the component_code and level_code were dropped from the dataframe as there was no variation in the parameters between 'normal' and 'anomaly' logs when compared manually. The 'Label' columns were converted to a binary format of 1s and 0s, with 0 indicating an anomaly. The 'blkID' column was also dropped as it would not be used in the LSTM. The final version of the dataframe is provided in table 9, it was exported as a csv and loaded into a separate python file, and imported into the LSTM.

Table 9. Reformatted dataframe inputted into the LSTM neuro net

	TagCode	LogLabel
0	2	1
1	4	1
2	8 7	1

The parameters for the LSTM neuro net were vocabulary_size, learning_rate, batch_size, num_epochs, the number of embedded dimensions, and the number of hidden dimensions. The default values for the parameters are summarized in table 10.

Table 10. Default parameters used by the LSTM

Parameters	Default Value
Vocabulary Size	22000
Learning Rate	1e-3
Batch Size	128
Num Epoch	25
Number of Embedded Dimensions	128
Number of Hidden Dimensions	256

The data was separated into training, validation, and testing sets at a ratio of 0.75, 0.05, and 0.2 for all tests conducted on the LSTM. Due to previous issues identifying anomalies, most of the logs were placed in the training set, however, due to the possibility of overfitting, 20 % of the data needed to be used for the training phase. Since the neuro net is only differentiating between logs that are and that are not anomalies, there are only two classes in the neuro net with 12,591 anomalies and 418,705 normal logs.

Due to the size of the data set and processing power requirements to process the data and wrangle the neuro net, all of the models were run using Google Colab Pro with a T4 and P100 GPU and 125 GB of Ram. Note only about 25 GB were required for processing the data.

EXPERIEMENT AND RESULTS

To gain insight into which method of machine learning would have the highest accuracy for anomaly detection and due to the large size of the dataset, different models were run using a subset of the real data with only 2,000 logs. These models were run using the "Level code", "Component code", and the "Tag code" columns.

These models were run in R and required the data to undergo variable reduction via Principal Component Analysis (PCA). The variable reduction was required due to the number of variables in our model. By removing unnecessary variables, it allowed the following models to focus on the components that were associated with most of the variance in the model. It was believed that the parameters with the largest variance would be associated anomalies in our data and could be used to identify anomalies after further analysis. The models that were then applied to the reduced data were Random Forest, Mixture Models

(Mclust), and a basic neuro net in R. These models were used because they are known as powerful models for machine learning with the first two being easy to implement [4].

The results of these models were disappointing with none of them being able to identify anomalies from the data provided, tables 11, 12 & 13. The model using random forest classified zero logs as anomalies and was subsequently run using the non-reduced data, producing the same result. Mclustering was only able to correctly identify 5 of the 68 anomalies and had 682 false positives, suggesting that this model is not properly differentiating between normal logs and anomalies. The basic neuro net also classified zero of the logs as anomalies. These poor results suggest that the data need to be wrangle differently or a more flexible model was needed.

Table 11. Classification Table – PCA & Random Forest

	Predicted Anomaly	Predicted Normal
Actual Anomaly	0	14
Actual Normal	0	365

Table 12. Anomaly Identification Count – PCA & Mclustering

Cluster	Actual Anomaly	Actual Normal	Anomaly/Normal Ratio
1	1	79	0.0127
2	1	102	0.0098
3	11	252	0.0437
4	13	217	0.0599
5	10	304	0.0329
6	4	204	0.0196
7	7	286	0.0245
8	20	375	0.0533
9	2	113	0.0177

Table 13. Classification Table – Basic Neuro Net

	Predicted Anomaly	Predicted Normal
Actual Anomaly	0	14
Actual Normal	0	365

To increase the flexibility of the model an LSTM was used with a count matrix based on the tag_codes associated with each block id. The LSTM was trained on the tag_codes as a string of numbers representing the tag_code of each block id. Splitting the data into training, validation, and testing data at a ratio of 0.75, 0.05, and 0.20. Using the following parameters, the accuracy of the model for each run is summarized in table 14.

Table 14. Summary of LSTM runs.

Run	Vocabulary Size	Learning Rate	Batch Size	Num of Epochs	Embedded Dimensions	Hidden Dimensions	Training Accuracy	Valid Accuracy	Testing Accuracy
1	22000	1e-3	128	25	128	256	97.6	97.5	97.3
2	22000	1e-3	128	25	220	440	99.13	99.08	99.03
3	22000	1e-4	128	25	220	440	99.13	99.08	99.03
4	22000	1e-5	128	25	220	440	99.12	99.07	99.01
5	22000	1e-2	128	25	220	440	99.11	99.06	99.02
6	22000	1e-4	128	25	230	460	99.14	99.08	99.04
7	22000	1e-4	128	25	300	600	98.86	98.78	98.64
8	22000	1e-4	128	25	200	400	99.02	98.97	98.93
9	22000	1e-4	128	20	230	460	99.14	99.08	99.04
10	22000	1e-4	128	30	230	460	99.14	99.08	99.04
11	24000	1e-4	128	25	230	460	99.15	99.03	98.98
12	20000	1e-4	128	25	230	460	99.13	99.10	99.03

Run 1 of the LSTM was run using default parameters and achieve a testing accuracy rate of 97.3%. The high training, accuracy, and testing rate indicates that the data and model are a good fit for identifying anomalies in the log files. The small variation between the testing and training accuracy indicates that the model is not overfitting to the data.

To optimize the LSTM, the complexity of the model was increased by increasing the number of embedded and hidden dimensions from 128 and 256 to 220 and 440, respectively. This increase in complexity allowed the LSTM to account for the complexity of our model. This in turn increased greatly the training and testing accuracy to 99.13% and 99.03% respectively. In runs 3 through 6 the learning rate was altered with a higher rate of 1e-2 and a lower learning rate of 1e-5 producing lower accuracy rates. A learning rate of 1e-4 had the same accuracy as run 2. Using a higher or smaller learning rate would decrease the accuracy of the model indicating that the LSTM reaches the global minima of the model with a learning rate between 1e-3 and 1e-4. To verify whether the number of dimensions could account for further complexity of the model the number of dimensions was altered in runs 6 – 8. These runs indicate that 230 and 460 embedded and hidden dimensions have the highest accuracy, but only slightly increase the test accuracy by 0.01 to 99.04. Due to the lack of change in the result no further alterations to the number of dimensions were made.

The vocabulary and the number of epochs were changed to determine whether the size of the data or altering the amount of time the LSTM had to train would alter the accuracy. These results proved minimal and did not increase the test accuracy. Therefore, the parameters outlined in table 15 produce the optimal results for our model with a training and testing accuracy of 99.14 % and 99.04 %, respectively.

Table 15. Parameters of the LSTM producing the highest training and testing accuracy.

Run	Vocabulary Size	Learning Rate	Batch Size	Num of Epochs	Embedded Dimensions	Hidden Dimensions	Training Accuracy	Valid Accuracy	Testing Accuracy
6	22000	1e-4	128	25	230	460	99.14	99.08	99.04

REFERENCES

[1] Cloudera. 2009. Apache Hadoop Log Files: Where to find them in CDH, and what info they contain. (September 2009). Retrieved May 4, 2021 from https://www.facebook.com/notes/cloudera/apache-hadoop-log-files-where-to-find-them-in-cdh-and-what-info-they-contain/134161942002/

[2] Zhao, Z., Xu, C. & Li, B. A LSTM-Based Anomaly Detection Model for Log Analysis. *J Sign Process Syst* (2021). https://doi.org/10.1007/s11265-021-01644-4

[3] Lakshmi G Mandagondi. February 2021. *Anomaly Detection in Log Files Using Machine Learning Techniques.* Master's Thesis. Blekinge Institute of Technology, Karlskrona, Sweden.

[4] Lu, Siyang. Detecting Anomalies From Big Data System Logs. 2019. *Electronic Theses and Dissertations* 6525 (2019). https://stars.library.ucf.edu/etd/6525

[5] Christopher Olah. 2015. Understanding LSTM Networks. (August 2001). Retrieved May 4, 2021 from https://colah.github.io/posts/2015-08-Understanding-LSTMs/

[6] Rafal Jozefowicz, Wojciech Zaremba, and Ilya Sutskever. 2015. An empirical exploration of recurrent network architectures. In Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37(ICML'15). JMLR.org, 2342–2350.

[7] Amit Singh Rathore. 2020. LSTM — Introduction in simple words. (September 2020).). Retrieved May 4, 2021 from https://medium.com/nerd-for-tech/lstm-introduction-in-simple-words-fe544a45f1e7

[8] Amir Farzad & T. Aaron Gulliver. 2021. Log Message Anomaly Detection and Classification Using Auto-B/LSTM and Auto-GRU. arXiv: 1911.08744. Retrieved from https://arxiv.org/abs/1911.08744