# HDFS Data Log Analysis Report
# Data 586

Mona Jia, Zhiyan Ma

## Abstract

Logs contain many useful information which are hard to read by humans. The logs represent the normality of the code status. By only looking at the logs, we can obtain the code status easily. We randomly choose part of the data from the HDFS logs 1 provided by a private cloud environment using benchmark workloads to reduce the capacity and calculation workload for the program. We matched the data with the anomaly label which assigned each block id a status. After merging the label with the sample logs data, we assign each block a type label. Then separated the data into training and testing data sets to perform a perceptron learning on our data. The perceptron procedure is used to predict the normality of our logs. Each training set is trained based on a multi-layer perceptron and predicts the normal or abnormal for the data. The analysis firstly used the linear calculation of the perceptron. Then ReLu activation xor is involved in the analysis. After the machine learning analysis, the data are automatically separated into two sections. We use a scatter plot to show the normal with a cross and abnormal with a square in the scatter plot. To make sure the result is reliable and accurate, we use a decision tree to cluster the data and compare it with our perceptron model.

## Introduction

We are using the data that comes from the Hadoop Distributed File System (HDFS) generated logs. HDFS is an Apache Hadoop service. It uses multiple computers as the network to solve complicated problems involving massive amounts of data and

computation. HDFS is a distributed file-system that stores data on commodity machines, providing very high aggregate bandwidth across the cluster. The HDFS system has been more and more popular those years.

The logs were generated by a private cloud environment using benchmark workload. The data also contains a manually input block id, normality paris. Each line of the log is sliced into traces by using the block id.

As we studied the machine learning methods in our master studying especially the data586 class, we decided to use what we've learnt from the class to find the rich sources and potential information form those logs. The logs are hard to read by humans and seem irregular. However, the log contains all information related to the network hooks up, failure reasons, results and successful message. The logs free programmers from printing every single line of their code. If we can detect the potential information stated by the logs, the debugging and troubleshooting process will be way easier. The code will also be much cleaner without those printing statements. Investigation and understanding the information potentially contained by the logs are critical and meaningful for the development of the code. Furthermore, in the future maintenance and improvement of the code, many software engineers may not be able to reproduce the same environment or cannot understand where the bug was. The logs information is a perfect pitch to tell the engineers where the problem is. It contains all labels, information, and the failure reasons. Simply looking at the last line of the logs which states the latest error is not enough for a large program maintenance and developing any more. The development of complication and interactivation between codes makes maintenance harder and harder. Thus, it is essential to analyze and understand the potential information contained in the logs.

## Literature Review

We read the literature of "Detecting Large-Scale System Problems by Mining Console Logs" and tried to find some thoughts from it. The passage illustrates that there was lots of information buried in the logs. Even though the logs seem a mess, they are pretty organized with a certain pattern with identifiers and stat

vars. All of those structures lead to a certain message in the log. All messages are strongly correlated with each other which provides us much more information for the code. In the report they have, they mentioned that they are parsing the log into a message and variable pairs which inspired us in our method too. Then they choose a feature creation to link the message pair with a certain feature. After that they label each trace of log with a normal or abnormal label. At the end they are using an image to visualize their result.

They are using 22 widely developed web systems to generate 2 tables of data. In the report, they mention the main parts are made by loggings. Different languages show different patterns in logs. However, even those logs are made by different programming languages, they are following the same pattern. They processed the data on over 20 EC2 nodes which generates 24 millions lines of logs. In our analysis, we cannot adapt this big size of the data. To have a reasonable run time and processing time, we only parse a part of the data and organize it into the HDFS sample csv file.

Due to the language differences, each log shows different content which is hard to parse. We met the same problem during our analysis. They have separated the whole parsing process into two steps: a static source code analysis step and the runtime log parsing step. They first transfer all logs into strings then parsing based on their index query.

Based on the parsing data, they created two categories for the data, a state ratio vector and a message count vector. The state ratio vector is able to capture the aggregated behavior of the system over a time window. The message count vector helps detect problems related to individual operations. Based on those two ratios, they further have more analysis. In the end, they decided to use Principal Component Analysis to balance term weighting with information retrieval. They are trying to use the distance between each log to detect the anomalies. At the end of the experiment, they are using the accuracy calculation and visual representation to prove their result.

# Background

After reading the literature, we adopt many aspects from the article and also make lots of improvements on our own. We combined what we learnt in the data 586 class and the method taught in the literature. We adopt partial of the original data produced with the network and processing. We also have a similar parsing which assigns the label for our graph. We did not use the PCA but the Perceptron and ReLu. To prove our result, we also calculated the accuracy and compared our PCA result with a decision tree clustering.

# Methodology

## i Data pre-processing

After adopting the logs data set and the anomaly label csv provided by the HDFS log analysis github, we use jupyter notebook and panda library in python to create two data frames for two csv files. We then parse the sample logs file into three categories, block id, block length and block type. The figure below has shown an example of our parsing data. For the abnormal label file, we translate the anomaly to 1 and normal to 0 for our future analysis and visualization. Then we merge the source data and label data together to create a label for each line of logs. Then we create a set for each unique value in the data type list. We randomly sign a number for each category. Since we are plotting the scatter plot for our perceptron and ReLu, we need both x-axis and y-axis. The translated block type will be our axis.

The data type we finally got:

```
['dfs.DataNode$PacketResponder:',
 'dfs.FSNamesystem:',
 'dfs.DataNode$DataXceiver:',
 'dfs.DataBlockScanner:',
 'dfs.FSDataset:',
 'dfs.DataNode:']
```

*Figure 2*

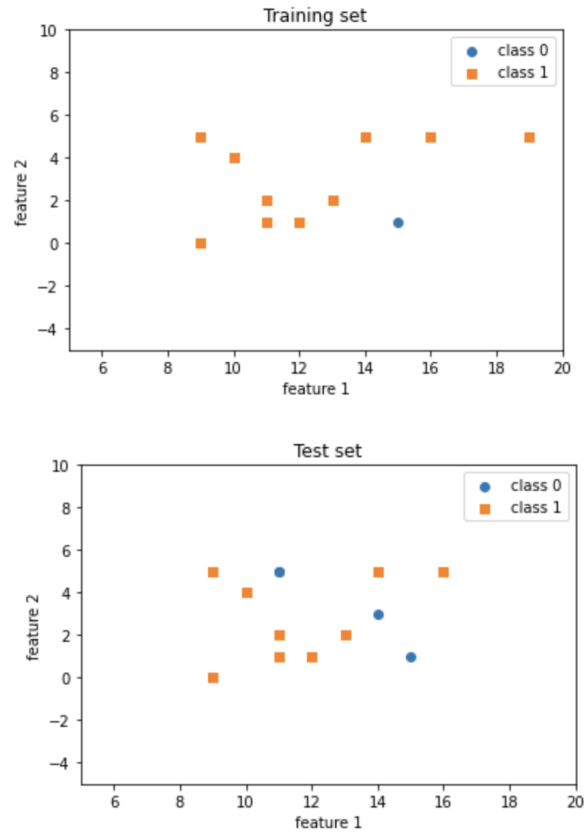
## ii Perceptron

After merging the label with the sample data, we got a pair of axes which can represent each point.

Then we separate the data into a training set and a testing set with a

| | BlockId | BlockLength | BlockType |
|---|---|---|---|
| 0 | blk_38865049064139660 | 11 | dfs.DataNode$PacketResponder: |
| 1 | blk_-6952295868487656571 | 11 | dfs.DataNode PacketResponder: |
| 4 | blk_-6670958622368987959 | 11 | dfs.DataNode$PacketResponder: |

proportion of fifty-fifty. Then we draw all points in the scatter plot with the features we got from the source sample csv and the label csv. The color depends on their normality. A blue “o” represents the normal status and an orange “square” represents the abnormal. Here is the original plot for the training set and test set we have for all nodes.




After separating the data into two data sets, we perform a perceptron analysis.

We adopt the perceptron model from the previous class code and make some adjustments based on our data.

The perceptron model uses a XOR gate to control the training data to predict the result. It assigns each of our data a certain weight and minus the bias. Then predict a 0 or 1 for the final result. In our research, the 0 represents the normal and 1 shows the abnormality of the log status. We use a forward method to predict a result then use a backward method to calculate the error. In each loop of training we also reshape the weight and re-calculate the bias to achieve a better prediction. Here is the algorithm for our Perceptron model.

---

```
class Perceptron():
    def __init__(self, num_features):
        self.num_features = num_features
        self.weights = torch.zeros(num_features, 1,
                                   dtype=torch.float32,
device=DEVICE)
        self.bias = torch.zeros(1, dtype=torch.float32,
device=DEVICE)

        # placeholder vectors so they don't need to be
recreated each time
        self.ones = torch.ones(1)
        self.zeros = torch.zeros(1)

    def forward(self, x):
        linear = torch.add(torch.mm(x, self.weights),
self.bias)
        predictions = torch.where(linear > 0.,
self.ones, self.zeros)
        return predictions

    def backward(self, x, y):
```

```
        predictions = self.forward(x)
        errors = y - predictions
        return errors

    def train(self, x, y, epochs):
        for e in range(epochs):
            for i in range(y.shape[0]):
                # use view because backward expects a
matrix (i.e., 2D tensor)
                errors = self.backward(x[i].reshape(1,
self.num_features), y[i]).reshape(-1)
                self.weights += (errors *
x[i]).reshape(self.num_features, 1)
                self.bias += errors

    def evaluate(self, x, y):
        predictions = self.forward(x).reshape(-1)
        accuracy = torch.sum(predictions ==
y).float() / y.shape[0]
        return accuracy
```

---

To prove our result is accurate, we calculate the accuracy rates. The test rate prediction accuracy reaches 96.5%. It means that 96.5% chance of our prediction of the abnormality is the same as the real life situation.

## iii Decision Tree

To double check the prediction of our results, we also calculate the clustering with the decision tree model. The unsupervised decision tree clustering provided by the tree library also proves our prediction with an accuracy rate of 96.60%. From both accuracy rate calculations we can conclude that our perceptron model has accurately predicted the normality of the logs.

## iv MLP Linear and ML ReLu for visualization

To advance analyzing our model, we want to color the area and draw a more advanced scatter plot. We then use the multiple layer linear perceptron model.

---

```
class MLPLinear(torch.nn.Module):

    def __init__(self, num_features, num_hidden_1,
num_classes):
        super(MLPLinear, self).__init__()
        self.num_classes = num_classes
        self.linear_1 = torch.nn.Linear(num_features,
num_hidden_1)
        self.linear_out =
torch.nn.Linear(num_hidden_1, num_classes)

    def forward(self, x):
        out = self.linear_1(x)
        logits = self.linear_out(out)
        probas = F.softmax(logits, dim=1)
        return logits, probas
```

---

We include hidden layers in our perceptron model to further analyze our result. We also include a rectified linear activation function (ReLu) activation function.

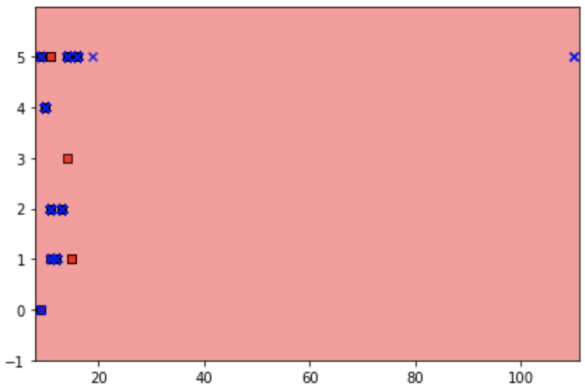
```
-----------------------------------------------
class MLPReLU(torch.nn.Module):

    ##Insert your code here
    def __init__(self, num_features, num_hidden_1,
num_classes):
        super(MLPReLU, self).__init__()
        self.num_classes = num_classes
        self.linear_1 = torch.nn.Linear(num_features,
num_hidden_1)
        self.linear_out =
torch.nn.Linear(num_hidden_1, num_classes)

    def forward(self, x):

        out = self.linear_1(x)
        out = F.relu(out)
        logits = self.linear_out(out)
        probas = F.softmax(logits, dim=1)
        return logits, probas
-----------------------------------------------
```

By using those two activation functions, we can classify our nodes with the color. The final classification plot we have is shown below.




# Experiment and Result

Before we implemented the Perceptron with the Multiple Linear Perceptron and ReLu activation function, we applied a neural network for our mode. Unfortunately, we lost all of our original code due to the computer crush. All files were lost after the update to the latest MacOs software. Under the stress of the deadline, we changed from a more complicated CNN model to an easier perceptron learning. We were also struggling on which model we needed to use at the beginning of the project. We wanted to apply as much of the algorithm we learnt from data586 as possible to our project.

At last, we choose the Perceptron model to train and predict the normality of the data. Then use a decision tree to prove the accuracy of our model. At the end we used a multi-layer perceptron and ReLu to draw the clustering of the nodes. The accuracy of our perceptron shows a 96.5% when predicting based on a fifty-fifty training-testing data set. The decision tree has proven a 96.0% accuracy for our perceptron model.

Our model can clearly predict the normality of each log which will provide significant improvements in the future code maintenance. The

programmer can use our model to analyze the anomaly of their logs. It means that they do not need to read every single line of logs or print out the result of their code to make sure the code is running smoothly. By using our model, programmers can save more time in maintenance too.

# Reference


- Shilin He, Jieming Zhu, Pinjia He, Michael R. Lyu. Loghub: A Large Collection of System Log Datasets towards Automated Log Analytics. *Arxiv*, 2020.
- Wei Xu, Ling Huang, Armando Fox, David Patterson, Michael Jordan. Detecting Large-Scale System Problems by Mining Console Logs, in Proc. of the 22nd ACM Symposium on Operating Systems Principles (SOSP), 2009.