System log analysis for the detection of anomalies through Machine Learning techniques

1. Introduction

Logs are one of the main methods used in software development to collect information at runtime about the running system. Often, these are used in server architectures to keep track of the operations performed within it, facilitating maintenance operations in the event of software failures or defects.
In recent years, the large amount of logs produced by the servers has made it increasingly difficult to manually analyze and resolve problems from them.
To solve this problem, therefore, a group of researchers has tried to develop artificial intelligence techniques capable of automatically analyzing the logs: these studies have shown that making use of these techniques allows us to simplify the work of analysis by extracting information. most important on the behavior of the servers.

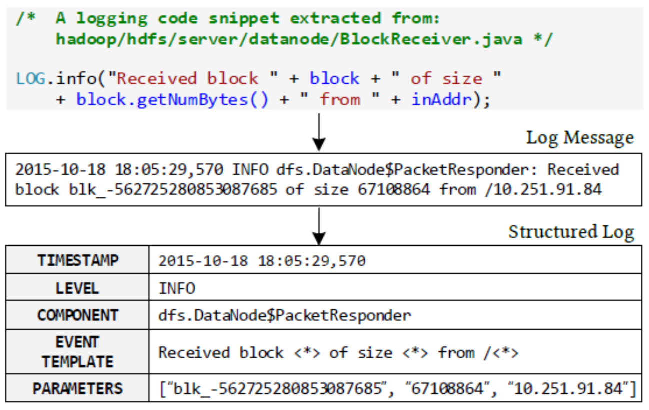
In this report we will consider the work done by the LogPAI group of researchers (http://www.logpai.com); the product presented by the team is divided into three main parts:
- Loghub: a vast collection of datasets to be able to carry out the analysis activities;
- Logparser: a set of techniques for converting logs;
- Loglizer: a set of analysis techniques for detecting anomalies in the logs.

1.1 Loghub (https://github.com/logpai/loghub)

As mentioned before, Loghub collects several datasets that can be analyzed for research purposes.
Among these we find:
- **Hadoop Distributed File System:**
  - **HDFS_1** it is generated in a 203 node HDFS. Comes with a file where all anomalies have been manually tagged.
    Logs are divided into sequences based on block_id: each sequence is then associated with the correct label (normal / anomaly).
  - **HDFS_2,** instead, it is collected by aggregating the logs from the HDFS cluster present at the team's laboratory in CUHK.
- **Hadoop:** a big data processing framework that allows the processing of very large datasets through clusters, making use of simple programming models. Logs were collected by injecting three different errors during the simulation. Labels are not provided.
- **BGL:** a dataset of logs collected by a BlueGene/L supercomputer at the LLNL. Logs contain alert messages identified by certain tags.

1.2 Logparser (https://github.com/logpai/logparser)

Logparser offers a set of techniques for automatic log conversion.
One of the fundamental aspects of this analysis is to be able to standardize the log structure to be able to generalize the analysis techniques of the next phase.
All the converters in the library produce the same result, making use of different algorithms.
Among the best we have: **Drain, IPLoM, LenMa** e **AEL.**


```
/*  A logging code snippet extracted from:
    hadoop/hdfs/server/datanode/BlockReceiver.java */

LOG.info("Received block " + block + " of size "
    + block.getNumBytes() + " from " + inAddr);
```

```
2015-10-18 18:05:29,570 INFO dfs.DataNode$PacketResponder: Received
block blk_-562725280853087685 of size 67108864 from /10.251.91.84
```

TIMESTAMP	2015-10-18 18:05:29,570
LEVEL	INFO
COMPONENT	dfs.DataNode$PacketResponder
EVENT TEMPLATE	Received block <*> of size <*> from /<*>
PARAMETERS	["blk_-562725280853087685", "67108864", "10.251.91.84"]

1.3 Loglizer (https://github.com/logpai/loglizer)

Loglizer is a collector of different models for log analysis. This part deals with overriding the best known classification techniques present in various libraries (eg sklearn).
According to what the team has produced, the best performing techniques are: **Decision Tree, SVM** e **LR**.

2. Proposed model

2.1 Loghub

The model proposed in this report will focus on the **HDFS_1** dataset, as it is already complete with all the tools to carry out a precise analysis through the labels provided by Loghub.
The log structure of HDFS_1 is like:

```
081109 203615 148 INFO dfs.DataNode$PacketResponder: PacketResponder 1 for block
blk_38865049064139660 terminating
```

Logs are therefore structured as **<Date> <Time> <Pid> <Level> <Component>: <Content>.**
Each log is associated with a blk_id which aggregates more logs to the same reference block.

As regards the labels, each blk_id is associated with a label in the file anomaly_label.csv, like:

```
blk_-1608999687919862906,Normal
blk_7503483334202473044,Normal
blk_-3544583377289625738,Anomaly
blk_-9073992586687739851,Normal
```

2.2 Logparser

Moving on to the second phase, the parser used is **Drain**.
Drain collects the logs within a csv file, characterizing them by an EventTemplate, an EventId and a ParameterList.
The result we get is of the type:

```
1,081109,203615,148,INFO,dfs.DataNode$PacketResponder,PacketResponder 1 for block
blk_38865049064139660 terminating,dc2c74b7,PacketResponder <*> for block <*>
terminating,"['1', 'blk_38865049064139660']"
```

This characterization is carried out to group the different types of logs and allow the analyzers to classify them correctly.

2.3 Loglizer

Before proceeding to the analysis phase, a DataFrame is built starting from the csv generated by the Logparser.
First, all events linked to the same blk_id are collected:

BlockId	**EventSequence**
blk_-1608999687919862906	[E5, E22, E5, E5, E11, E11, E9, E9, E11, E9, E…]
blk_7503483334202473044	[E5, E5, E22, E5, E11, E9, E11, E9, E11, E9, E…]

Subsequently, we build a matrix in which we put in the columns all the generated eventId and on the rows the blk_id of the logs being analyzed. In the intersections between blk_id and eventId we insert the number of times the blk_id is connected to the relative eventId:

	E22	E5	E26	E11	E9	E2	E7	E10	E14	E3	E6	E16	E18	E25
blk_-1608999687919862906	1	3	3	3	0	0	0	0	0	0	0	0	0	0
blk_7503483334202473044	1	3	3	3	0	1	0	0	0	0	0	0	0	0

At this point, a normalization of the data is carried out using the algorithms (in this case tf-idf and zero-mean) and the resulting DataFrame is passed to the chosen analyzers.

The implementation of new analyzers can be done by creating a class that exports the fit, predict and evaluate methods; an example of implementation:

```
def __init__(self, attributes):
    self.classifier = Classifier(attributes)

def fit(self, X, y):
    self.classifier.fit(X, y)

def predict(self, X):
    y_pred = self.classifier.predict(X)
    return y_pred

def evaluate(self, X, y_true):
    y_pred = self.predict(X)
    precision, recall, f1 = metrics(y_pred, y_true)
    return precision, recall, f1
```

In our case **Multi Layer Perceptron** and **Nearest Neighbors** were also added to the available analyzers.

3. Results

Once the matrix of values corresponding to the logs blocks has been generated, the data is analyzed using the techniques present within LogLizer plus some added later.
The results obtained from our experiment were obtained using a subset of 750 thousand logs (~ 100 MB) of the HDFS_1 dataset and the related labels.
The data is divided linearly, associating half of the subset to the Train and half to the Test, in this subdivision:

Total:	103804 instances	2419 anomalies	101385 normal
Train:	51901 instances	**1209 anomalies**	50692 normal
Test:	51903 instances	**1210 anomalies**	50693 normal

All techniques were applied with default parameters.

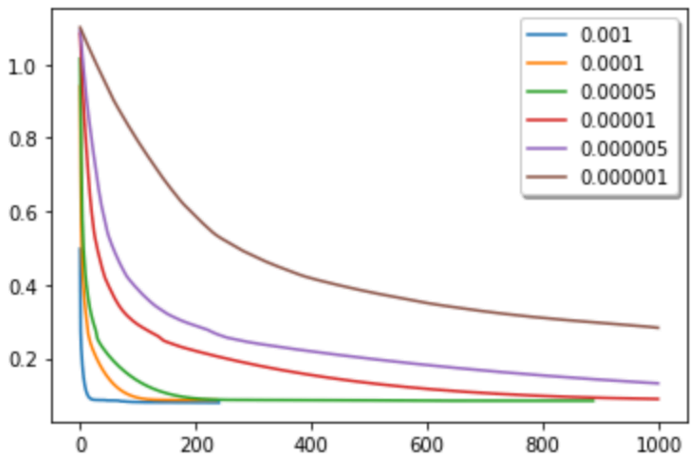
TRAIN:

	Decision Tree	Invariant Miner	Isolation Forest	Log Clustering	Logistic Regression	PCA	SVM	MLP	Nearest Neighbors
Precision	1	0,397213	0,239745	1	1	0,376623	1	1	1
Recall	0,259719	0,188586	0,280397	0,259719	0,259719	0,167907	0,259719	0,259719	0,254756
F1	0,412344	0,255748	0,258483	0,412344	0,412344	0,232265	0,412344	0,412344	0,406065

TEST:

	Decision Tree	Invariant Miner	Isolation Forest	Log Clustering	Logistic Regression	PCA	SVM	MLP	Nearest Neighbors
Precision	0,052419	0,044843	0,014973	0.010046	0.048893	0.041818	0.044944	0.367347	0.367347
Recall	0,032231	0,049587	0.361157	0.109091	0.043802	0.038017	0.049587	0.014876	0.014876
F1	0,039918	0,047096	0.028754	0.018397	0.046207	0.039827	0.047151	0.028594	0.028594

In detail, the **MLP** technique is analyzed with solver **adam**.
First let's see how it behaves when the learning rate changes.
As can be seen from the graph and the table of results, the best learning rate, among those considered, is equal to 0.00001.



L-Rate	Test Precision	Test Recall	Test F1	Train Precision	Train Recall	Train F1
0.001	1.0	0.2597	0.4113	0.36	0.0149	0.0286
0.0001	1.0	0.2597	0.4113	0.0524	0.0322	0.0399
0.00005	1.0	0.2597	0.4113	0.0524	0.0322	0.0399
0.00001	1.0	0.2597	0.4113	0.36	0.0149	0.0286
0.000005	1.0	0.2589	0.4113	0.3673	0.0149	0.0286
0.000001	0.983193	0.0968	0.1762	0.8	0.0066	0.0131

Therefore, having chosen the learning rate (1e-5) and the tolerance (1e-9), we varied alpha to verify which performed better:

Alpha	Test Precision	Test Recall	Test F1	Train Precision	Train Recall	Train F1
0.1	1.0	0.2597	0.4123	0.3673	0.0149	0.0286
0.001	1.0	0.2597	0.4123	0.0524	0.0322	0.0399
0.00001	1.0	0.2597	0.4123	0.3673	0.0149	0.0286
0.0000001	1.0	0.2597	0.4123	0.3673	0.0149	0.0286
1E-09	1.0	0.2597	0.4123	0.3673	0.0149	0.0286

And finally we tried standardizing the data using StandardScaler:

Alpha	Test Precision	Test Recall	Test F1	Train Precision	Train Recall	Train F1
0.1	1.0	0.2597	0.4123	0	0	0
0.001	1.0	0.1671	0.2863	0	0	0
0.00001	1.0	0.1671	0.2863	1	0.0066	0.0131
0.0000001	1.0	0.1671	0.2863	1	0.0066	0.0131
1E-09	1.0	0.1671	0.2863	1	0.0066	0.0131