Assignment #2 – Log Analysis for Anomaly Detection

Yoke Kai Wen, 2020280598

November 11, 2020

1 Introduction

Anomaly detection is important in the management of large-scale distributed systems. Logs that record detailed runtime information are widely used for anomaly detection. However, checking logs manually is impossible due to challenges posed by unstructured texts, huge amount of logs and diverse types of logs. Hence, we need automated log-based anomaly detection methods. The framework of automatic log anomaly detection typically consists of four steps: (1) log collection, (2) log parsing, (3) feature extraction, (3) anomaly detection. In Part 1 of this assignment, I will explore step (2) using 4 log parsing methods; in Part 2 of this assignment, I will explore steps 3 and 4 using two unsupervised anomaly detection methods.

2 Part 1: Comparing current log parsing methods

The purpose of log parsing is to extract a group of event templates from unstructured logs. The DSN paper [1] introduces many methods for automated log parsing, and the authors provided a toolkit (https://github.com/logpai/logparser) for the log parsing algorithms and also test log files which were used for this assignment.

2.1 Brief overview of log parsing algorithms

The log parsing algorithms can be divided into two groups: heuristics based (SLCT, IPLoM) and clustering based (LKE, LogSig).

2.1.1 SLCT - Simple Logfile Clustering Tool

SLCT works by constructing a word vocabulary of word frequency and position which are used to generate candidate log templates. The main tunable parameter is `support_threshold` - (word, position) with frequency exceeding the `support_threshold` are selected for candidate construction.

2.1.2 IPLoM - Iterative Partitioning Log Mining

IPLoM first partitions log messages into different clusters based on their lengths, then for each cluster, it further partitions each log message at positions with the least number of unique words. It then performs further partition by searching for mapping relationships between two token positions selected using a heuristic criterion, and finally generates log templates from each cluster. The main tunable parameters are `lowerBound` and `CT` (clustering goodness threshold) associated with the heuristic criterion used by IPLoM.

2.1.3 LKE - Log Key Extraction

LKE first clusters raw log messages using hierarchical clustering algorithms and then further split up clusters based on heuristic rules, and finally generating log templates from every cluster. The main tunable parameter `split threshold` is associated with the heuristic rules used by LKE.

2.1.4 LogSig

LogSig first converts each log message into a set of word-position pairs, and based on these word pairs, LogSig calculates a potential value for each log message that is used to determine which cluster each log message belongs to. After a number of iterations, the clusters are determined, and then log templates for each cluster is generated. The main tunable parameter is `groupNum` which determines how many clusters are formed in LogSig.

2.2 Methodology

The toolkit provides five types of logs (BGL, HDFS, HPC, Proxifier, Zookeeper), each consisting of 2000 logs, and the groundtruth structured logs are also provided for evaluating the parsing algorithms. On these five log types, we test four log parsing algorithms: LogSig, IPLoM, SLCT and LKE, also available in the toolkit.

2.3 Choice of optimum parameters

I mainly used the benchmark parameters recommended by the authors which are specific for each log dataset, because after some tries of parameter tuning I could not achieve significant performance improvement. I also included preprocessing for every algorithm and log dataset by setting the recommended regex for each experiment as I found that it improved performance significantly for most algorithms compared to having no regex. The parameters chosen are shown in Table 1. Other parameters, if not mentioned, are the default in the toolkit. Furthermore in the paper [1], the authors mentioned that parameter turning, especially for clustering-based parsing methods is time-consuming (see Figure 1 for algorithm runtimes), therefore since the benchmark parameters yield decent performance, I used them for evaluation of each algorithm on each log dataset.

	BGL	**HPC**	**HDFS**	**Zookeeper**	**Proxifier**
SLCT	support: 6 regex: Y	support: 7 regex: Y	support: 120 regex: Y	support: 10 regex: Y	support: 8 regex: Y
IPLoM	CT: 0.4 lowerBound: 0.01 regex: Y	CT: 0.58 lowerBound: 0.25 regex: Y	CT: 0.35 lowerBound: 0.25 regex: Y	CT: 0.4 lowerBound: 0.7 regex: Y	CT: 0.9 lowerBound: 0.25 regex: Y
LKE	split_threshold: 30 regex: Y	split_threshold: 10 regex: Y	split_threshold: 3 regex: Y	split_threshold: 20 regex: Y	split_threshold: 3 regex: Y
LogSig	groupNum: 500 regex: Y	groupNum: 800 regex: Y	groupNum: 15 regex: Y	groupNum: 46 regex: Y	groupNum: 10 regex: Y

Table 1: Table showing optimal parameters chosen for each parsing algorithm and log dataset

2.4 Bar charts for performance evaluation

2.4.1 Runtimes

Figure 1 shows the runtimes for each parsing algorithm and log dataset. As expected, the clustering algorithms take significantly longer than the heuristics based algorithms. LogSig took

much longer for BGL and HPC compared to the other datasets, while LKE is more consistent across different datasets but took significantly longer for Proxifier. IPLoM is the most efficient for all datasets. This is because SLCT and IPLoM scales linearly with the number of log messages. Although LogSig also scales linearly with the number of log messages, its running time also scales linearly with the number of events, hence BGL and HPC which had more event types took longer to parse. The time complexity of LKE is $O(n^2)$, so it has long runtimes. LKE takes especially long for Proxifier which has only 8 event types in the ground truth, but LKE instead generated a template with 49 events for it.

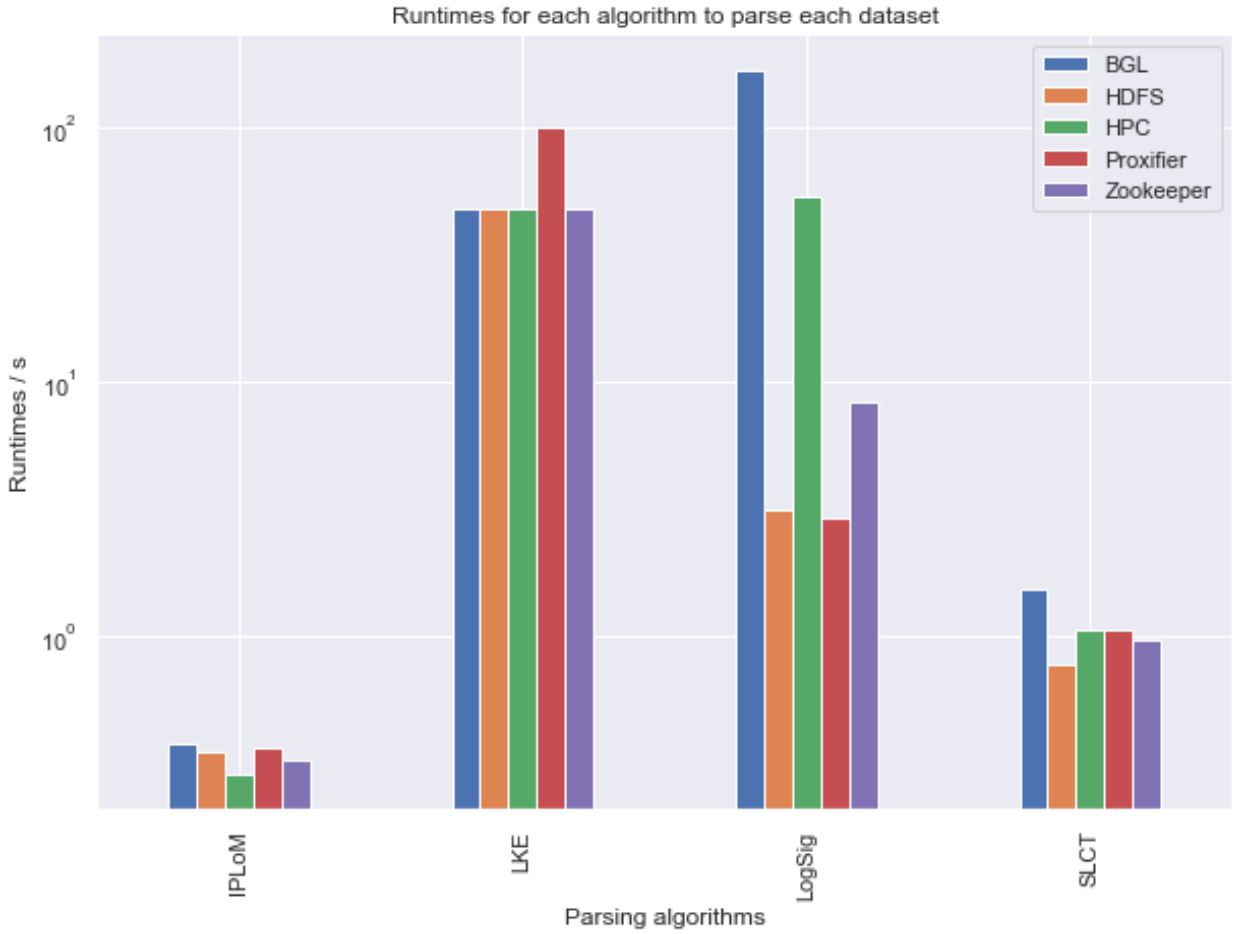

Figure 1: Runtimes for each parsing algorithm on each log dataset

2.4.2 RandIndex

Figure 2 shows the RandIndex measure for each parsing algorithm and log dataset. RandIndex is calculated by $RI = \frac{TP+TN}{TP+FP+FN+TN}$. A true positive (TP) decision assigns two similar documents to the same cluster; a true negative (TN) decision assigns two dissimilar documents to different clusters; a false positive (FP) decision assigns two dissimilar documents to the same cluster; a false negative (FN) decision assigns two similar documents to different clusters. Therefore, the RandIndex simply measures the percentage of decisions that are correct.

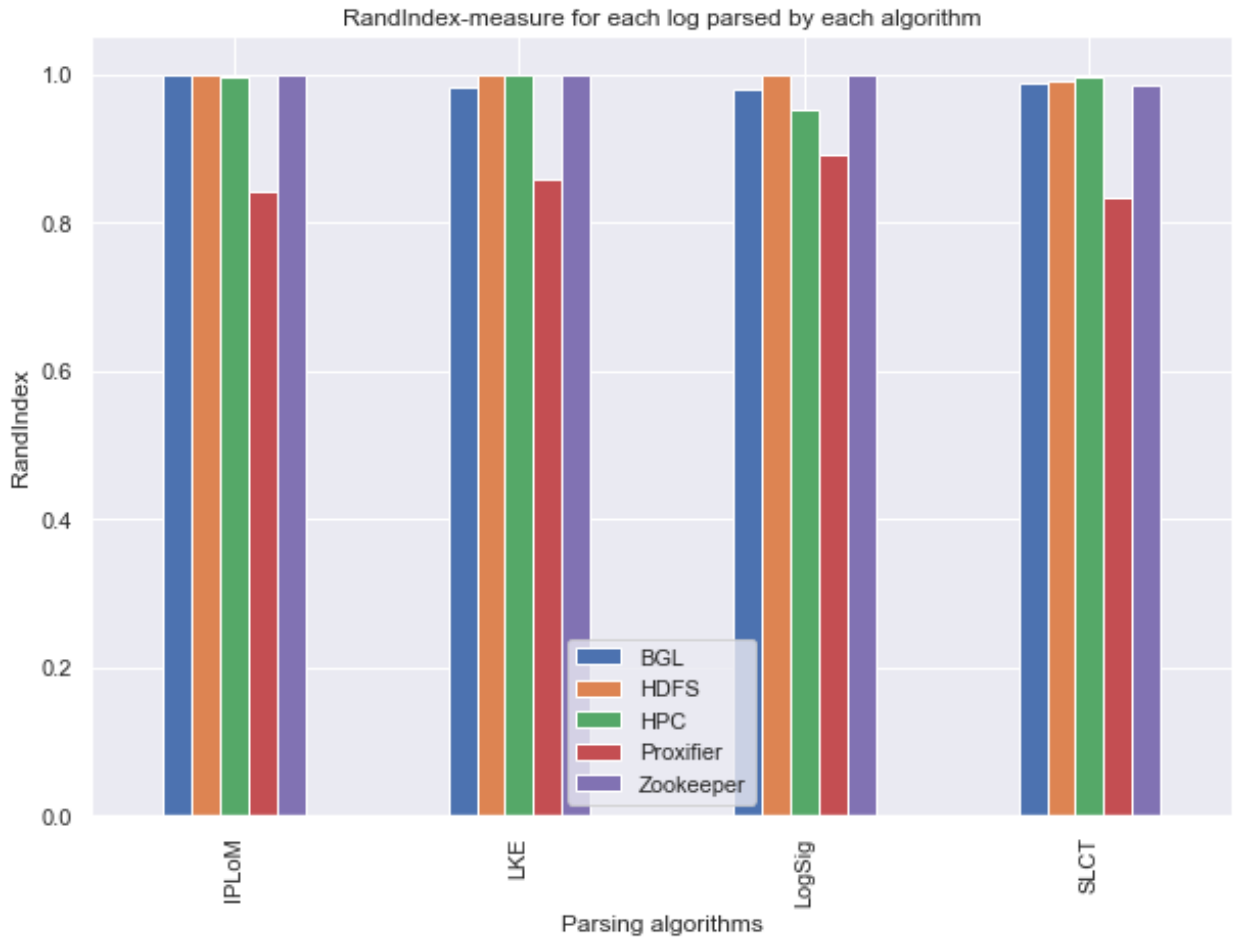

Figure 2: RandIndex-measure for each parsed algorithm on each log dataset

The structured logs generated by each algorithm is compared with the ground truth structured logs to determine the RandIndex. Overall, all parsing algorithms achieve quite high accuracy on most log types, except for Proxifier, which has very few events in the ground truth template, and most of the parsing algorithms mistakenly discovered more event types. IPLoM has the best performance overall.

2.4.3 F1-measure

Figure 3 shows the F1-measure for each parsing algorithm and log dataset. F1-measure is calculated by $F_1 = \frac{(\beta^2+1)\times Precision\times Recall}{\beta^2\times Precision+Recall}$, where $\beta = 2$ here. F1-measure is different from RandIndex in that RandIndex gives equal weights to false positives and false negatives, but since separating similar documents is sometimes worse than putting dissimilar documents in the same cluster, F1 measure can penalise false negatives more strongly than false positives by selecting $\beta > 1$.

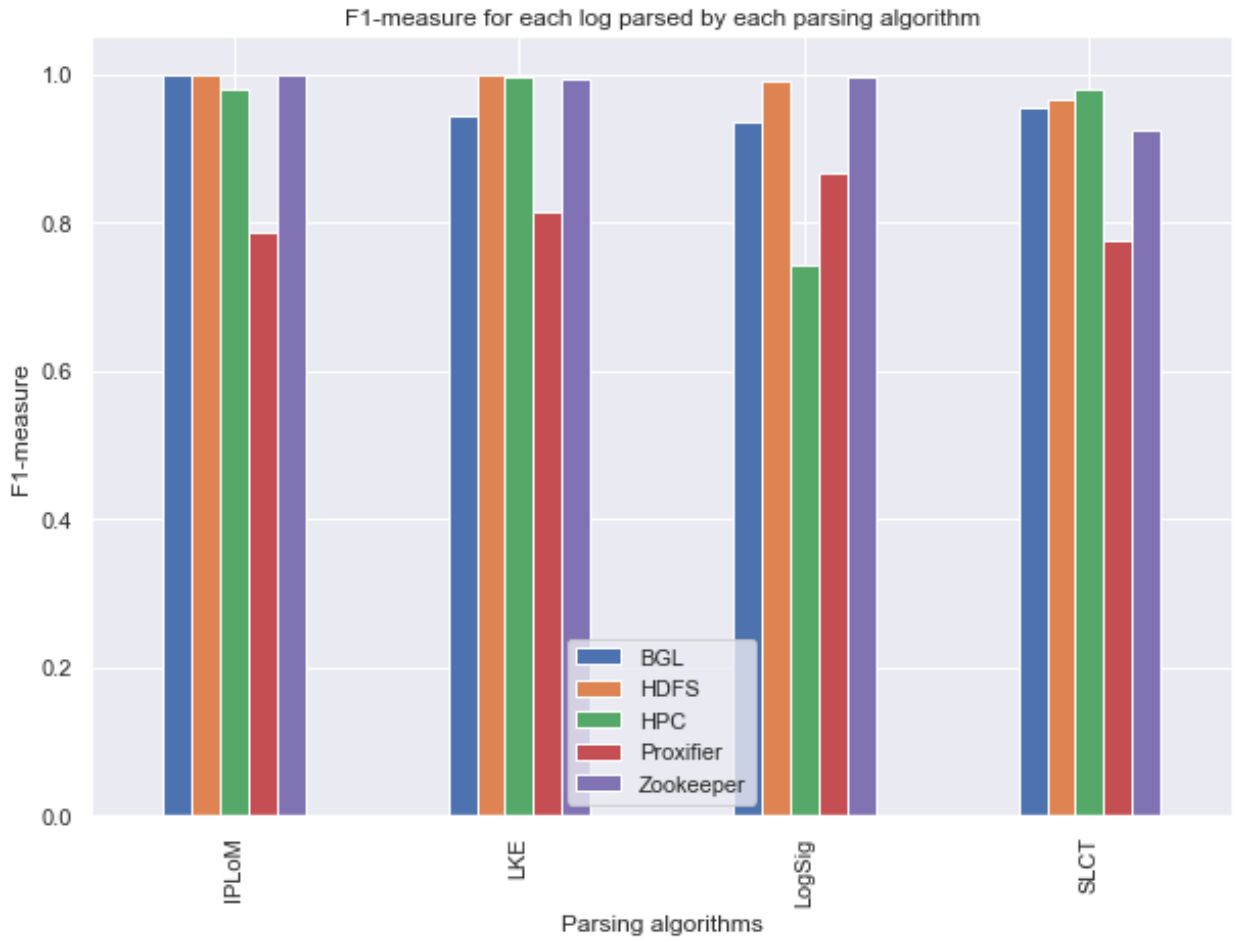

Figure 3: F1-measure for each parsed algorithm on each log dataset

F1-measure performance is largely the same as RandIndex performance, except that we see that for F1, when false negatives are penalised more heavily, LogSig now performs much more poorly for HPC than indicated by its RandIndex score. SLCT also performs more more poorly for Zookeeper now. On the other hand, IPLoM remains consistent in performance, suggesting that it has a low false negative rate, and still remains as the best parsing algorithm.

2.5 Experience and insights

After replicating the procedure of the paper [1], I find that most of my results are consistent with the paper's findings: the four log parsing methods achieve high overall parsing accuracy, simple log preprocessing using regex can improve log parsing accuracy, clustering-based log parsing algorithms are slow and parameter tuning is time-consuming. In addition, the measure used to evaluate parsing performance is also important, for example, F1-measure with the appropriate β is much more insightful than just vanilla RandIndex. Parsing performance is also very dependent on log type, for example the number of event types and the length of events. IPLoM performs the most robustly for almost all log types, while LKE and LogSig (clustering algorithms) are heavily dependent on log type. Finally, parsing performance is also very sensitive to tuning parameters.

3 Part 2: Comparing anomaly detection methods

The ISSRE paper [2] introduces many approaches for anomaly detection on structured logs. In this assignment, I explore the unsupervised learning approaches and implement two of them using the toolkit provided by the authors at https://github.com/logpai/loglizer. We are provided with a `HDFS.log` file of roughly 11 million log messages with roughly 575k unique block ids, together with a `Label.csv` files that contains the anomaly labels for the log messages. `HDFS.log` is parsed using IPLoM with regex, and then the parsed output is used for anomaly detection with PCA and Invariants Miner algorithms.

3.1 Feature Extraction

Before the anomaly detection models can run on the structured parsed logs, the structured logs have to be converted into an input matrix of (windows x events) counting the occurrence of events in each window. Typically, we window the logs in terms of timestamps, but for `HDFS.log`, we have `block_id` for each log message that records the allocation, writing, replication, deletion of certain blocks. Hence, we use session windows instead with each window representing a `block_id`, and count the number of events occurring in each block. After the input matrix is formed, preprocessing is applied to it such as applying normalisation, depending on the requirements of the anomaly detection models.

3.2 Brief overview of unsupervised anomaly detection models

I learnt about three unsupervised anomaly detection models from the paper [2]: PCA, Invariants Mining, and Log Clustering.

3.2.1 PCA - Principal Component Analysis

PCA projects the high dimension event count matrix (n x e) onto k principal components, where k is calculated by finding components which capture the most variance along the high-dimension data. The main parameter tunable here is `n_components`, which is a float setting the variance ratio that the k principal components should cover. The span of these k principal components form the normal subspace, while the anomaly subspace is formed by the remaining (n-k) components. Then, the projection vector of an event count vector to the anomaly subspace is calculated. If the magnitude of the projection vector exceeds a certain threshold, the corresponding event count vector will be reported as anomaly. The main parameter tunable here is `c_alpha` which is used to calculate the anomaly detection threshold using Q-statistics.

3.2.2 Invariants Miner

The Invariants Miner model (IM) mines the linear relationships between each event that occurs in the program, which are known as program invariants. It works because logs that have the same `block_id` often represent the execution flow of that session, for instance a file must be closed after it was opened, hence the number of events of file opening and that of file closing should be the same for a particular `block_id`. Therefore, if the number of events do not follow the expected linear relationships in a `block_id`, we can mark it as an anomaly.
Thus, IM first estimates the invariant space using singular value decomposition, which determines the amount r of invariants (linear relationships) that need to be mined. Then, it finds out

the invariants by a brute force search algorithm. In the toolkit provided, IM only mines relationships involving two variables to reduce search complexity. Finally, each mined invariant candidate is validated by comparing its support with a threshold. This is associated with the main tunable parameters `percentage` which is the proportion of samples required to satisfy an invariant candidate, and `epsilon` which is the threshold for estimating the invariant space. During anomaly detection, a new log sequence is checked if it obeys the invariants, and will be reported as an anomaly if at least one invariant is broken.

3.2.3 Log Clustering

Log Clustering requires two training phases: knowledge base initialisation phase and online learning phase. In the first phase, the event count vectors undergo agglomerative hierachical clustering which generates two sets of vector clusters - normal and abnormal as knowledge base. A representative vector for each cluster is calculated by computing the centroid. In the second phase, the clusters formed are further adjusted by adding event count vectors one by one. If the smallest distance from each cluster is less than a threshold the event count vector will be added to the nearest cluster, and then the representative vector updated, otherwise a new (anomalous) cluster is created. Subsequently, Log Clustering can be deployed to detect anomalies by finding the closest cluster that each new event vector belongs to.

3.3 Methodology

For this assignment, I ran PCA and IM on the IPLoM parsed HDFS log. Log Clustering was not run because it has too high computational complexity. The log is uniformly split into train and test dataset distributions are shown in Table 2.

Train (80%)			**Test (20%)**		
Total	**Anomaly**	**Normal**	**Total**	**Anomaly**	**Normal**
460048	13470	446578	115013	3368	111645

Table 2: Table showing train-test anomaly distribution

3.4 Choice of optimum parameters

I had to personally tune several parameters as the `HDFS.log` file was not part of the toolkit and hence benchmark parameters were not available. The most relevant parameters I have were those for `HDFS_100k.log_structured.csv` provided in the toolkit, so I started off with those parameters, and adjusted dataloading parameters, preprocessing parameters and internal parameters in each anomaly detection model until I received decent accuracy. The optimal parameters I found are shown in Table 3. The parameters specified are those that have a significant effect on anomaly detection accuracy, other parameters that are not stated explicitly simply use the default settings in the toolkit. Note that I did not do an exhaustive grid search due to time limitations.

	Dataloading	Preprocessing	Internal
PCA	train_ratio: 0.8 split_type: 'uniform'	term-weighting: tf-idf normalization: zero-mean	c_alpha: 3.2905 num_components: 0.9
IM	train_ratio: 0.8 split_type: 'uniform'	None	percentage: 0.98 epsilon: 0.4

Table 3: Table showing optimal parameters for anomaly detection models

3.5 Bar charts for anomaly detection model evaluation

3.5.1 Runtime

Figure 4 shows the model fitting, train evaluation and test evaluation runtimes for PCA and IM models. For model training, PCA takes 0.1s while IM takes a whopping 893 seconds with an 80-20 train-test split. Prediction runtimes on train and test data are negligible, and in fact it is surprising that PCA takes longer for prediction than for training, while IM prediction is faster than that of PCA. Therefore, it is good that we limited IM to searching for invariants with two variables, instead of three or four variables, otherwise computational complexity would be far too high.

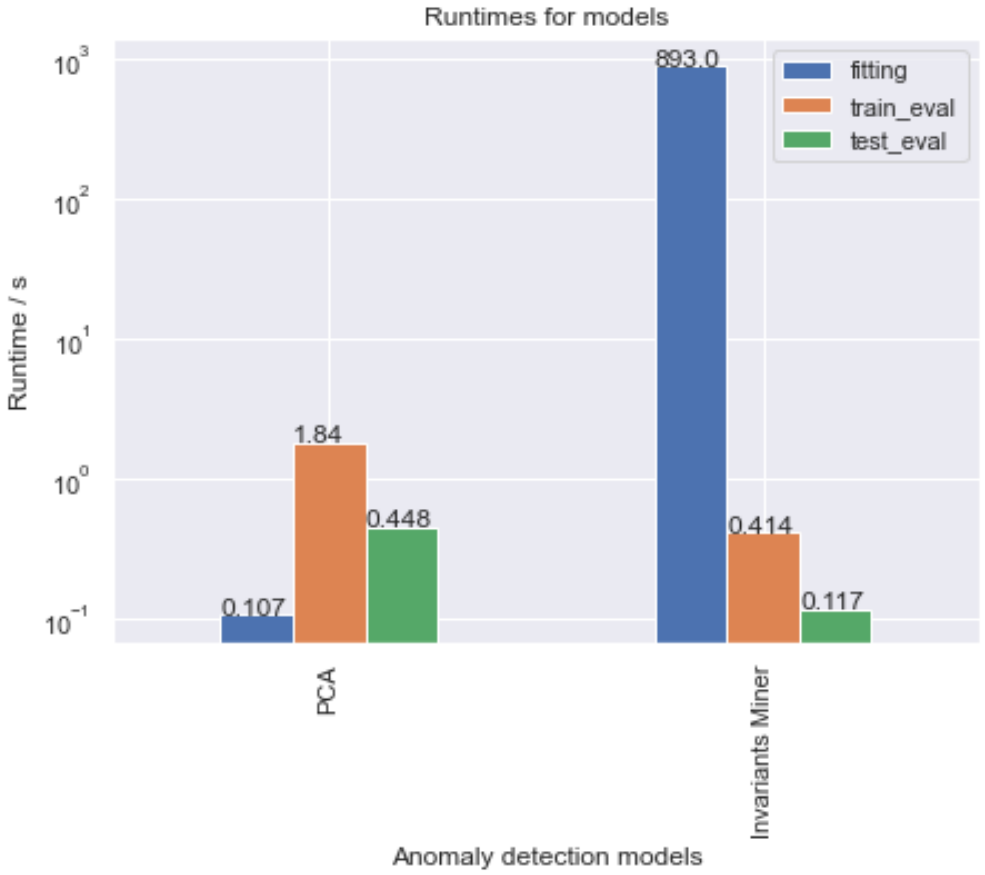

Figure 4: Fitting, train evaluation and test evaluation runtimes for anomaly detection models

3.5.2 Precision score

Figure 5 shows the precision scores of PCA and IM on train and test data logs. PCA has higher precision scores than IM, however this might not be significant, since precision merely measures the total number of true anomalies out of those predicted, and it is possible that a lot of true anomalies remain unspotted (large number of false negatives).

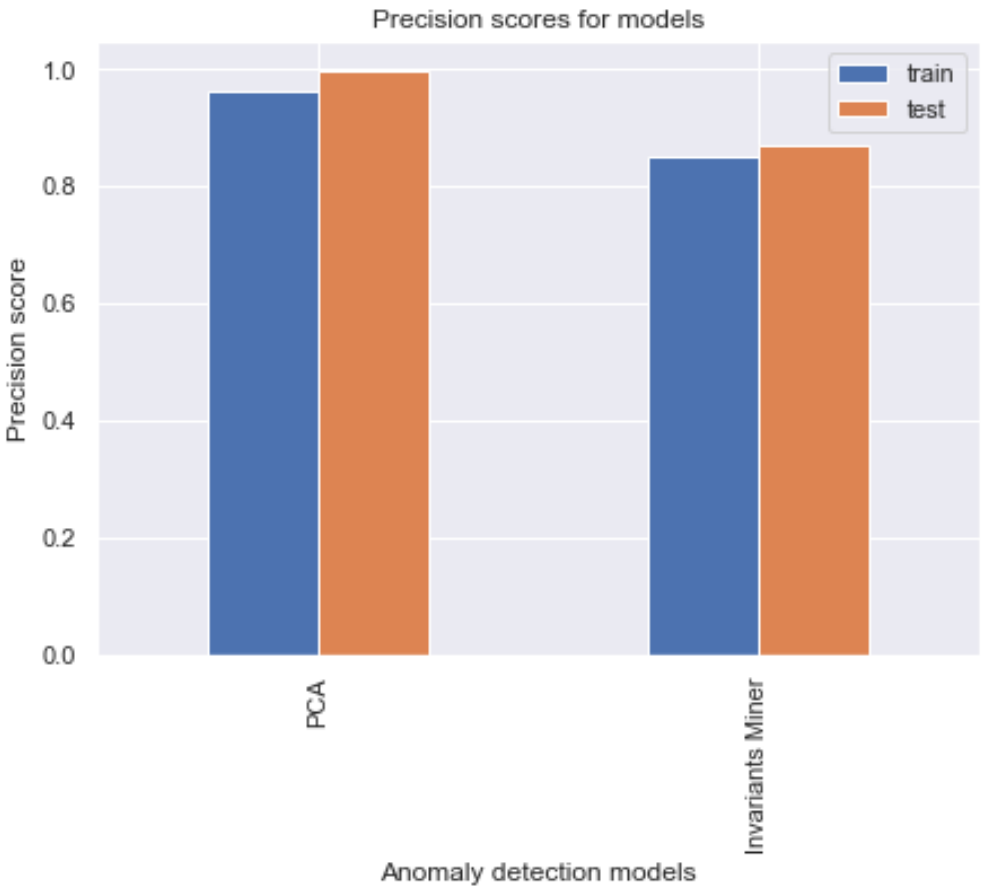

Figure 5: Train and test precision scores for anomaly detection models

3.5.3 Recall score

Figure 6 shows the recall scores of PCA and IM on train and test data logs. IM has perfect recall while PCA has roughly 70% recall. This means that IM is able to pick out 100% of the true anomalies, which is quite impressive. However, again, recall score is not telling enough, because it is possible that IM picks out many points as anomalies leading to many false positives which would be bad.

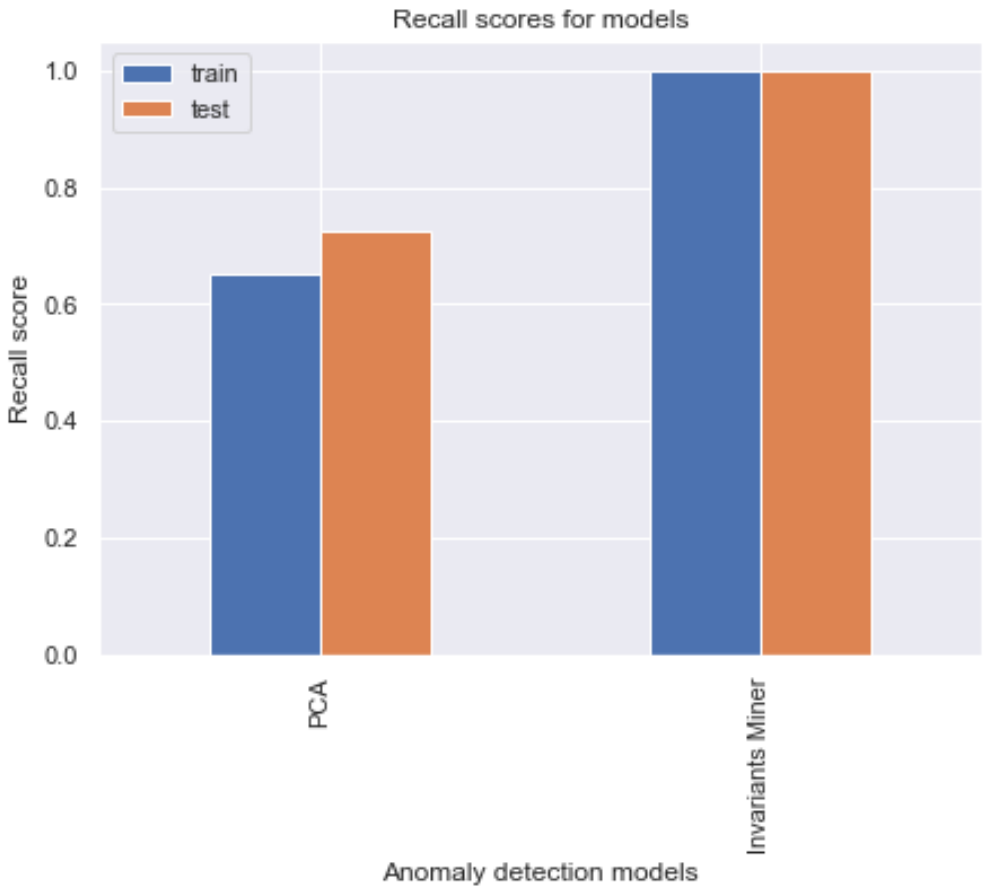

Figure 6: Train and test recall scores for anomaly detection models

3.5.4 F1 score

Figure 7 shows the F1-measure scores for PCA and IM models when predicting on the train and test datasets. Overall, IM performs better than PCA, with test F1-score exceeding 80%. Surprisingly, test score is higher than training score for both models. Perhaps it just happens that the test set contains more obvious anomalies than the train set.

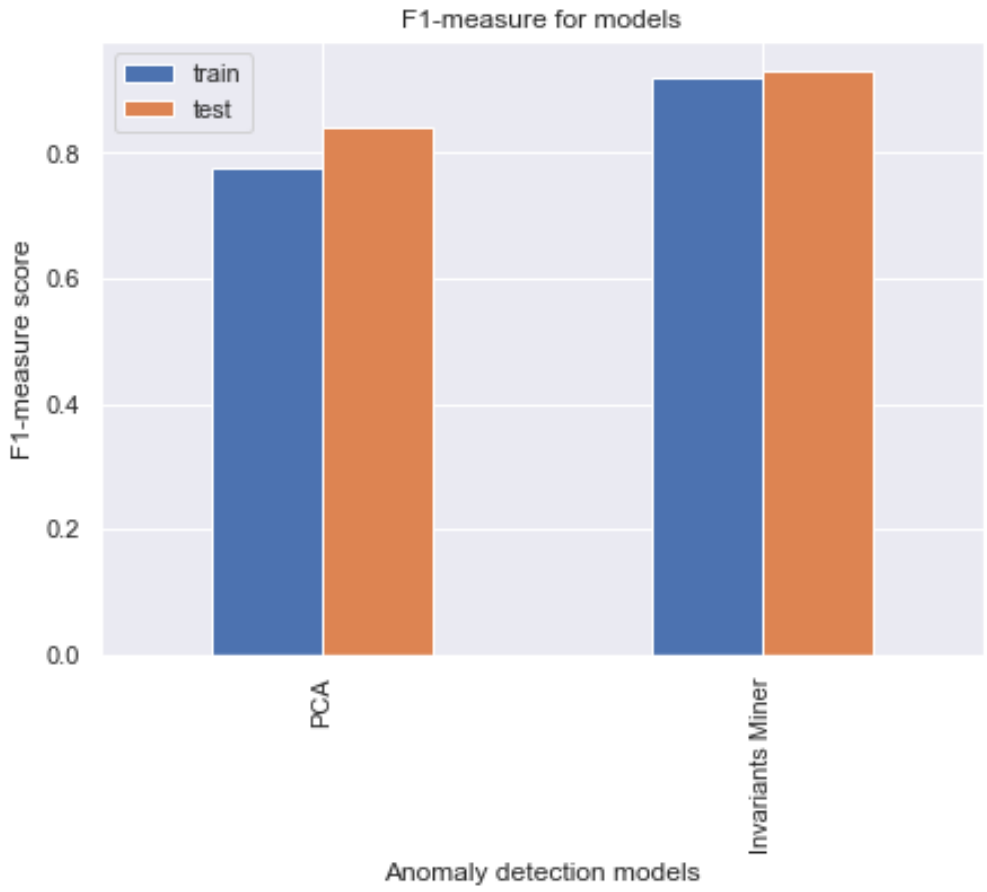

Figure 7: Train and test F1-measure for anomaly detection models

3.6 Three relationships mined by Invariants Miner

After fitting Invariants Miner on the training logs, the code outputs a series of relationships in the format shown in Figure 8.

```
====== Model summary ======
Invariant space dimension: 39
Mined 39 invariants: {(0, 1): [1.0, -3.0], (0, 2): [-1.0, 1.0], (0, 3): [1.0, -1.0], (0, 4): [1.0, -1.0], (5, 6): [-1.0, 1.0]
```

Figure 8: Some of the mined invariants printed by IM model

The indices in the round brackets correspond to a particular events, and the coefficients in the square brackets correspond to the coefficients in the linear relationships. For instance, (0,2):[-1.0, 1.0] is the relationship: $-1.0 \times n(event_0) + 1.0 \times n(event_2) = 0$. Table 4 shows the mapping between some of the event indices and the actual event description in the parsed template.

Event index	Event ID	Event template
0	*adb8aa1f*	Receiving block src / dest /
2	*d4e8dd53*	BLOCK* NameSystem.addStoredBlock blockMap updated is added to size <*>
3	*5f061fc2*	PacketResponder <*>for block <*>
4	*9d7e930c*	Received block of size <*>from /

Table 4: Table matching event index output by IM to event ID and description in parsed template

Observing the invariants output in Figure 8, three relationships are: (1) $n(event_0) = n(event_2)$; (2) $n(event_0) = n(event_3)$; (3) $n(event_0) = n(event_4)$. Matching the event indices to the actual event templates in the parsed log template as shown in Table 4, we realise that the number of all four events should be equal to each other. This makes sense because when a new block is received ($event_0$), the metadata about stored blocks needs to be updated ($event_2$). Similarly when receiving a new block ($event_0$), communication between the source and destination is required thus invoking the event for packet responder ($event_3$). In the same vein, the process of receiving a block ($event_0$) would be followed by the event of acknowledging that a block has been received ($event_4$). Hence, this is a good example of how the IM model is very explainable.

3.7 Experience and insights

Part 2 was significantly more challenging than Part 1. One reason was because the `HDFS.log` was not part of the toolkit, and additional effort has to be made for parsing and parameter tuning.

3.7.1 Parsing of large logs

Parsing itself was quite troublesome, because `HDFS.log` contained a large amount of logs, so initially I failed to fully parse everything at one go due to the lack of RAM. I considered splitting the entire log into smaller parts, and separately parsing, but the results were bad because the event templates from different splits did not match each other. Eventually, I found a way to get 25GB RAM on Google Colab, and managed to parse everything at one go, consuming 14.53GB RAM in the process. However, this method of parsing everything at one go seems very impractical in real-life situations when logs are in even larger volumes. The IPLoM code in the toolkit should ideally be modified to be able to parse logs in an online fashion.

3.7.2 Importance of train-test splits

Initially, the default setting was 50-50 train-test split, and I couldn't understand why after dataloading and preprocessing, the number of events (42) somehow decreased from the number reflected in the parsed templates (45). Then I realised if I increase the training dataset, the number of events after preprocessing increase. After some investigation, I found out that some events that occur rarely (e.g. frequency 3, 4, 5) could end up completely in the test set and not be seen at all in the training set, so the model is unaware of these events. In the end, I decided that this was not a big issue, since we are more interested in events that occur more frequently anyway. However, I did increase the train-test split to 80-20 so that the model can be exposed to a greater range of events and thus achieve better accuracy in anomaly detection.

3.7.3 Performance and explainability of anomaly detection models

Invariants Miner is very explainable, and helps us find out which invariant that an anomalous block id has broken, thus aiding us in finding the source of the problem. On the other hand, PCA is more opaque and we would have no idea why a block id is anomalous unless we go through further investigation. Furthermore, Invariants Miner is able to yield 100% recall with a reasonable rate of false positives, which makes it a very ideal and effective anomaly detection algorithm. Unfortunately, PCA misses a significant number of true anomalies and is far more inferior to Invariants Miner.

References

[1] P. He, J. Zhu, S. He, J. Li, and M. R. Lyu, "An evaluation study on log parsing and its use in log mining," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2016, pp. 654–661.

[2] S. He, J. Zhu, P. He, and M. R. Lyu, "Experience report: system log analysis for anomaly detection," in *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2016, pp. 207–218.