Log File Processing and Anomaly Detection on HDFS Log Dataset

Harpreet Kaur Guglani & Kristy Phipps

Abstract

Log files contain a wealth of information that is invaluable to the development and maintenance of many software systems. These files capture runtime details about a system during operation and can help developers improve performance. One way this is achieved is by using the log files to find and track atypical system behaviors and errors through a process of anomaly detection. While log files are a great resource to organizations, they are also inherently difficult to work with. Log files are unstructured and are produced in incredibly large volumes. To help combat some of these challenges, there has been a lot of great work done to provide open-source toolkits to support working with and extracting value and insights from log files. This paper explores a few of the tools created for undertaking this work, made available through the Logpai collection of repositories. This paper explores the use of these tools on a HDFS log file, which is just one of the many unstructured log files available on Loghub. Next, some of the tools within the Logparser and Loglizer libraries were explored to understand how log files are prepared and processed for analysis. Understanding that real-time detection of anomalies is one of the primary goals within industry, we focused our efforts on tools that process the data quickly and would provide highly accurate predictive results. For these reasons we moved forward with using the Drain parser from the Logparser library and decided to explore the Random Forests algorithm for prediction. Ultimately, this combination of tools worked incredibly well together, and resulted in a 99.98% prediction accuracy on the 15% test set withheld from the original data. We are hopeful this work will highlight both how easy to implement and how powerful these tools can be for processing unstructured log data into meaningful insights.

1. Introduction:

Logs serve as important records in today's increasingly digitalized world, recording detailed and valuable information about system runtime information. Log file processing and analysis therefore allows for insights into the health, performance, and operational status of various systems. In industry, logs have a wide variety of uses including supporting both operation and maintenance tasks, software development and developing overall system comprehensions [1]. Additionally, logs are used to help understand system usage analysis and for anomaly detection, performance modelling, duplicate issue identification, and failure diagnosis [2].

Often, these log files are the only resource available to organizations, developers, and engineers with this runtime information, which make log files an incredibly valuable resource to modern businesses. However, working with log files is a challenging pursuit on several fronts. Modern software systems and IT environments continue to increase in size and complexity which creates and compounds challenges, in addition to contributing to the ever-increasing volume of log files. Furthermore, logs are comprised of unstructured text, which adds complexity and challenge to working with this type of data.

Fortunately, there has been some great work and research completed in this field in recent years, and there is a wide variety of easily accessible tools and digestible literature on this subject. One excellent resource is the Logpai GitHub repository. Within this repository there are free publicly accessible system logs available for research purposes via Loghub [3]. Additionally, there are open-source toolkits for both log-parsing, and a machine learning-based toolkit that is specifically designed to assist with anomaly detection. This project and paper explore one of the labelled datasets available on Loghub repo – HDFS_1 (Hadoop Distributed File System) as well as some of the other tools available through the Logpai collection of repositories. Overall, the goal and purpose of this project was to develop an understanding and appreciation for the practice and tools of anomaly detection, while implementing a machine learning pipeline to achieve reasonable prediction accuracy in a supervised learning environment.

2. Background and Literature Review:

Anomaly detection is important in industry, as it plays a critical role in incident management by focussing on finding abnormal system behaviours in a timely manner. By detecting anomalies quickly, system developers and operators can find and resolve issues quickly, thus reducing system downtime [4]. In the past, this was accomplished via a combination of manual techniques, such as keyword searching, regular expressions or grok patterns, and domain knowledge [2].

Unfortunately, conventional approaches to anomaly detection are no longer adequate in large-scale situations for a variety of reasons. Perhaps the biggest challenge is the rate at which modern systems are generating logs. Estimates from 2013 stated that logs were being created at the rate of about 50 gigabytes per hour [5] and has likely only increased since then. Furthermore, with complex, large-scale parallel computing systems working to deliver services, it is highly likely that no one developer has a complete understanding of the entire system. Rather it is more likely that a developer has knowledge of a piece or sub-component of the system. This makes it particularly challenging to properly identify and diagnose issues that affect the entire system [4]. This is further compounded by the nature of the systems themselves, which often employ fault tolerance mechanisms to improve performance. This can include killing speculative tasks and running redundant tasks, which are not actually system failures or problems. Unfortunately, by employing manual keyword search process to identify issues, can lead to these being classified as anomalies when these are actually "false positive" results [4]. Given these challenges, it is not surprising that there is a large and growing demand for practical, automated log-based anomaly detection solutions.

In practice, there are four main steps to anomaly detection: log collection (data collection), log-parsing (data processing), feature extraction and finally anomaly detection [4].

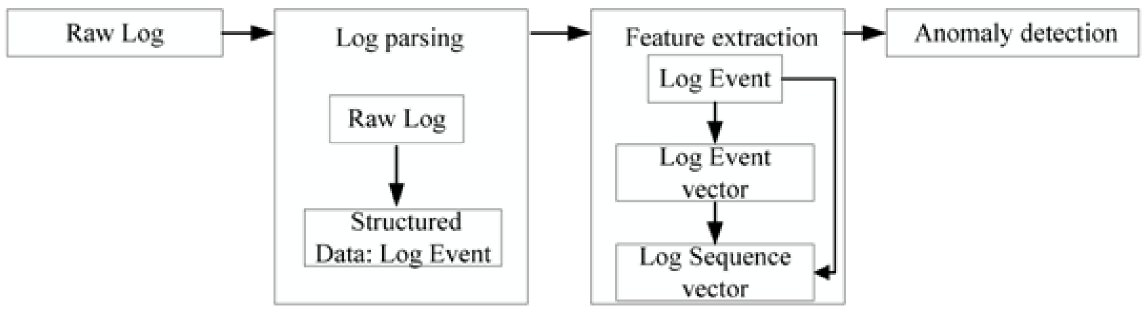
Log collection is generally straight-forward. Logs are generated at runtime and aggregated in a centralized storage space. This is often achieved via streaming data pipelines like Flume or Kafka. Once logs are collected and centralized, the challenge of log-parsing begins. Most logs consist of unstructured text, but this text contains a vast amount of interesting and valuable information. Most log files contain several forms of system runtime information, including date and time stamps, verbosity/severity levels, raw message content and identification information [2]. Thus, the first task in extracting this information requires processing this data into more meaningful and easily understood format. Generally, the parsing process splits the log message into two parts: a variable piece and a constant piece. The constant part contains information on the event type of interest and consists of plain text. The variable part on the other hand contains the runtime information and generally varies between events [6]. Ultimately, the goal of log-parsing is to transform the individual log message into specific events [7]. This gives the data a more structured format and allows for more efficient, filtering, searching, grouping, counting, mining and further processing of the log files.

Once the log-parsing process is complete and the logs have been separated into events, these events are then further processed into numerical vectors which are used for feature extraction. There are several effective feature extraction methodologies based on grouping techniques like fixed windows, sliding windows and session windows [4]. Once these windows are applied, each log sequence is first processed into an event count vector, and then combined to form a feature matrix [4]. Ultimately, it is this feature matrix which forms the structure on which machine learning models are applied. These models are then evaluated and tuned to detect anomalies present in the data. Once evaluated and tuned, these models can then be applied to new incoming logs to determine anomalies in the system in a timely fashion.

3. Methodology

In this section, we will provide more details on the specific tools and methods deployed during this project. For log parsing, we provide some of the basic ideas and introduce the specific log parser used. Next, feature extraction is discussed, which are applied on the parsed log events to generate feature vectors. After obtaining the feature vectors, we focus on presenting the anomaly detection approaches.

Figure 1:



3.1 Log Parsing

For this project, we focused on using and understanding some of tools available within the Logpai GitHub repository. One repo available there is the logparser library where there are thirteen different open source logparsers available for use. Each of these parsers has their own unique profile of strengths and weaknesses that are summarized in a benchmarking study [2]. A summary table from the benchmarking study is provided below (Figure 2) and shows that eight of the thirteen log parsers perform very well, with over 99% accuracy on the HDFS dataset [2]. In this review, two main log parsers IPLoM (Iterative Partitioning Log Mining) and Drain (a fixed depth tree based online log parsing method) had the most consistent accuracy ratings across all the various datasets (Figure 3) and thus were chosen for further investigation [2].

Figure 2:

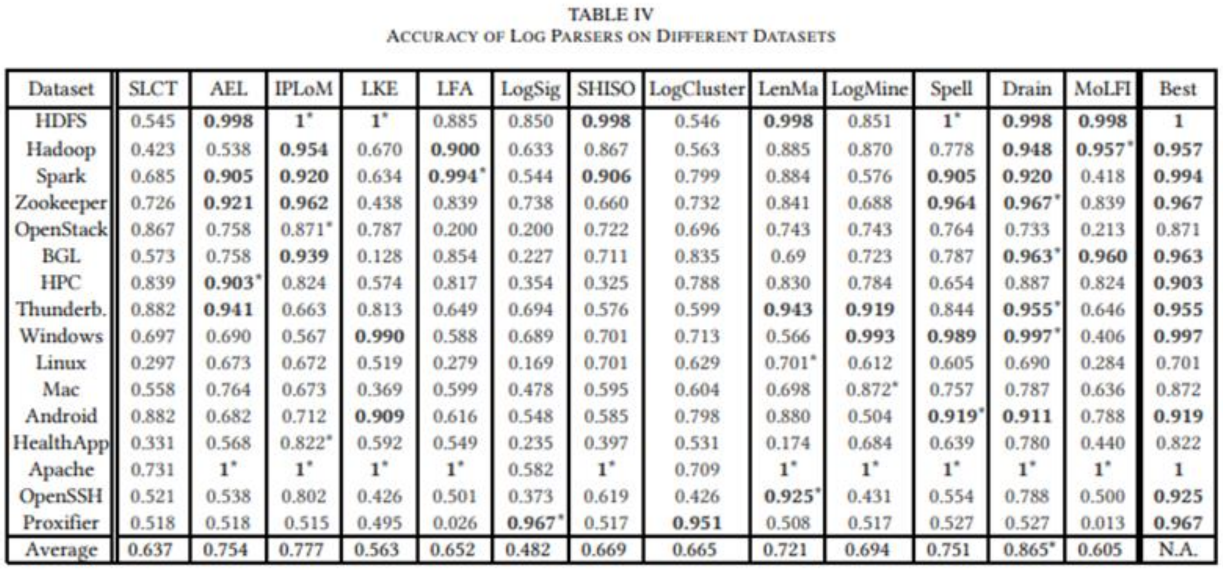


TABLE IV
ACCURACY OF LOG PARSERS ON DIFFERENT DATASETS

Dataset	SLCT	AEL	IPLoM	LKE	LFA	LogSig	SHISO	LogCluster	LenMa	LogMine	Spell	Drain	MoLFI	Best
HDFS	0.545	**0.998**	**1***	**1***	0.885	0.850	**0.998**	0.546	**0.998**	0.851	**1***	**0.998**	**0.998**	1
Hadoop	0.423	0.538	**0.954**	0.670	**0.900**	0.633	0.867	0.563	0.885	0.870	0.778	**0.948**	**0.957***	**0.957**
Spark	0.685	**0.905**	**0.920**	0.634	**0.994***	0.544	**0.906**	0.799	0.884	0.576	**0.905**	**0.920**	0.418	**0.994**
Zookeeper	0.726	**0.921**	**0.962**	0.438	0.839	0.738	0.660	0.732	0.841	0.688	**0.964**	**0.967***	0.839	**0.967**
OpenStack	0.867	0.758	0.871*	0.787	0.200	0.200	0.722	0.696	0.743	0.743	0.764	0.733	0.213	0.871
BGL	0.573	0.758	**0.939**	0.128	0.854	0.227	0.711	0.835	0.69	0.723	0.787	**0.963***	**0.960**	**0.963**
HPC	0.839	**0.903***	0.824	0.574	0.817	0.354	0.325	0.788	0.830	0.784	0.654	0.887	0.824	**0.903**
Thunderb.	0.882	**0.941**	0.663	0.813	0.649	0.694	0.576	0.599	**0.943**	**0.919**	0.844	**0.955***	0.646	**0.955**
Windows	0.697	0.690	0.567	**0.990**	0.588	0.689	0.701	0.713	0.566	**0.993**	**0.989**	**0.997***	0.406	**0.997**
Linux	0.297	0.673	0.672	0.519	0.279	0.169	0.701	0.629	0.701*	0.612	0.605	0.690	0.284	0.701
Mac	0.558	0.764	0.673	0.369	0.599	0.478	0.595	0.604	0.698	0.872*	0.757	0.787	0.636	0.872
Android	0.882	0.682	0.712	**0.909**	0.616	0.548	0.585	0.798	0.880	0.504	**0.919***	**0.911**	0.788	**0.919**
HealthApp	0.331	0.568	0.822*	0.592	0.549	0.235	0.397	0.531	0.174	0.684	0.639	0.780	0.440	0.822
Apache	0.731	**1***	**1***	**1***	**1***	0.582	**1***	0.709	**1***	**1***	**1***	**1***	**1***	**1**
OpenSSH	0.521	0.538	0.802	0.426	0.501	0.373	0.619	0.426	**0.925***	0.431	0.554	0.788	0.500	**0.925**
Proxifier	0.518	0.518	0.515	0.495	0.026	**0.967***	0.517	**0.951**	0.508	0.517	0.527	0.527	0.013	**0.967**
Average	0.637	0.754	0.777	0.563	0.652	0.482	0.669	0.665	0.721	0.694	0.751	0.865*	0.605	N.A.

Figure 3:

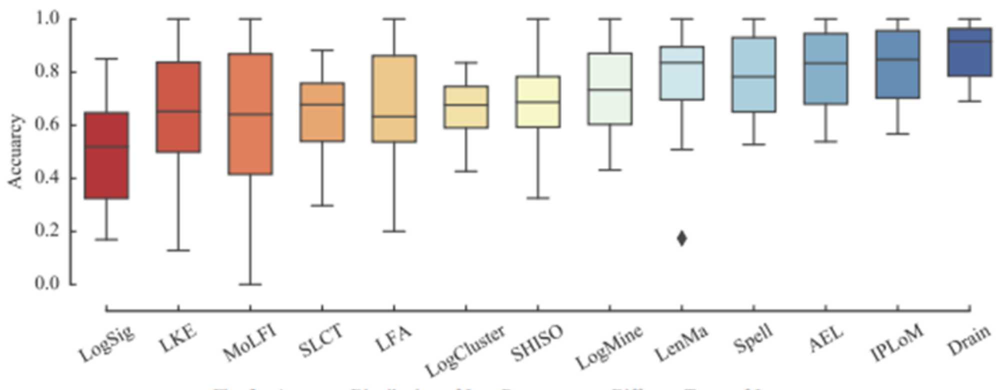


Fig. 2. Accuracy Distribution of Log Parsers across Different Types of Logs

A review of the IPLoM parser showed that it conducts log-parsing in a three-step hierarchical partitioning process to split the data into clusters before creating a log template. First, it partitions the logs into groups based on lengths. Secondly, within each partition the words at distinct positions are counted. Then log is then split at the position where there is the least number of unique words. Finally, further splits are made by searching for relationships between the unique tokens using a heuristic criterion [7]. At this point, the partitioning is complete, and each partition is considered a unique cluster and given a description and can be passed to other methods for further processing and analysis [8].

In comparing both IPLoM and Drain, some potential advantages to using Drain were discovered. The Drain parser performed very well on the HDFS data set and had highest average accuracy levels of all the logparsers examined in the benchmarking study. Additionally, Drain also has the advantage of being an online log parsing method. Most other methods, including IPLoM are designed to work offline and to process logs in batches [6]. Knowing that the high volume of logs creation is a major challenge, and that timely anomaly detection is highly desirable, Drain appeared to offer a more attractive solution to these challenges than some of its counterparts, including IPLoM. Drain achieves this by using a fixed depth parse tree and specially encoded rules to accelerate the parsing process and to allow for streaming processing of log files. Noting this ability, and the fact that Drain obtained a 51.85%~81.47% improvement in running time compared with a state-of-the-art online parser [6], made Drain the most attractive choice for this project.

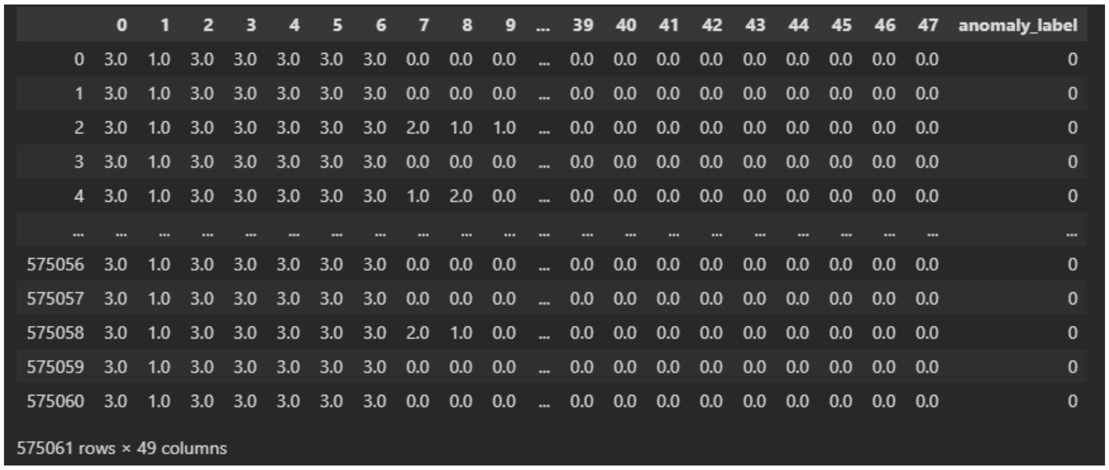
Drain follows a fixed depth tree-based algorithm which basically runs down a tree with the fixed depth length. In our implementation, we chose the depth to be four. This is important as it serves the purpose of searching through the group structures in the logs. Using simple regular expressions, combined with domain knowledge, Drain preprocesses the raw log files to remove common variables like IP address and block IDs [6]. Then Drain starts from the root node of the tree and compares each new log message with the log events already stored in the tree. Normally, this would be slow, but since Drain uses a fixed depth tree, this bounds the volume of groups and stops the tree from growing too large while also limiting the amount of searching the algorithm needs to complete [6]. The searching process includes determining the log message length and searching through the already existing groups and tokens for similarities [6]. For our study, we kept the similarity threshold at 0.5. In practice, this means that the Drain algorithm searches for a group based on the message of the particular log, and then matches the message to a log event which parallels to that log message based on the threshold 0.5. If it traverses the tree, and does not find a match, a new log group is created based on what the log message is trying to convey [6]. Finally, as a last step, the parser is updated appropriately to prepare for the next incoming log message.

3.2. Feature Extraction

One of the most important steps in the anomaly detection process is feature extraction. For this part, we used the loglizer library available within the broader Logpai GitHub repo [4]. This library provides a toolkit which uses machine-learning techniques for processing the parsed logs into separate events. This work involves encoding the events into numerical vectors. For instance, a feature vector of [3,1,2,0,0] means that Event 1 in the sequence occurred three times, while Event 2 only occurred once, Event 3 occurred two times, and Events 4 and 5 did not occur at all in this particular log sequence. Each log sequence is processed in an analogous manner, and eventually all the event count vectors are constructed in an event count matrix X. In this matrix, a given entry of $x_{i,j}$ records how many times the event *j* occurred in the *i*th log sequence with respect to each block id presented in the log data.

In using Drain in combination with the feature extractor available through the loglizer library, we obtained an event count matrix of 575061x 48 dimensions. This matrix represents the extracted events in an easy-to-understand integer format. We then added the known anomaly labels to the event matrix to form our final feature matrix which informs our predictive model.

Figure 4:



	0	1	2	3	4	5	6	7	8	9	...	39	40	41	42	43	44	45	46	47	anomaly_label
0	3.0	1.0	3.0	3.0	3.0	3.0	3.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0
1	3.0	1.0	3.0	3.0	3.0	3.0	3.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0
2	3.0	1.0	3.0	3.0	3.0	3.0	3.0	2.0	1.0	1.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0
3	3.0	1.0	3.0	3.0	3.0	3.0	3.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0
4	3.0	1.0	3.0	3.0	3.0	3.0	3.0	1.0	2.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
575056	3.0	1.0	3.0	3.0	3.0	3.0	3.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0
575057	3.0	1.0	3.0	3.0	3.0	3.0	3.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0
575058	3.0	1.0	3.0	3.0	3.0	3.0	3.0	2.0	1.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0
575059	3.0	1.0	3.0	3.0	3.0	3.0	3.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0
575060	3.0	1.0	3.0	3.0	3.0	3.0	3.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0

575061 rows × 49 columns

3.3 Modelling Framework and Approach

The loglizer feature extractor was used to create a matrix of predictors for a machine learning model. A model's framework uses these predictors as a way of representing the empirical relations between every aspect of inquiry when considered a scientific theory or research. Overall, forty-eight predictors were developed from the contents of the HDFS_1 dataset and used to inform our model. The predictors came in the form of a matrix which was divided into an 85/15 training/testing split. The idea behind using a training/testing split is that by withholding the test data throughout the model building and tuning process, the fitted model can be assessed against unseen data. This, then serves as a representation of how the model will perform on new unseen data.

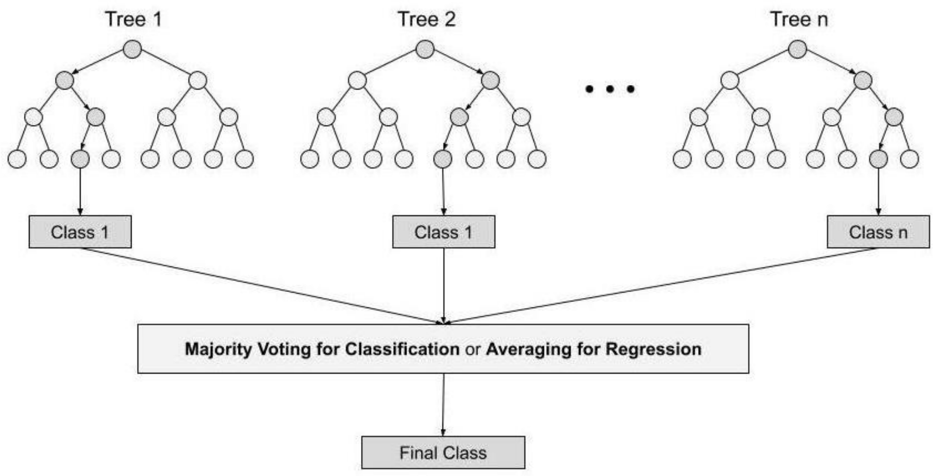
Ultimately, one of the most crucial steps in data analysis is the selection of the techniques which will become the model for the data. There is no correct or incorrect modelling method, but instead the selection depends entirely on the analysts to determine which method serves their purpose in the best way. For this project, we decided that our primary goal would be to attain a high level of predictive accuracy, rather than focusing on inference. Our understanding of anomaly detection in industry is that accurate, real-time prediction of anomalies is desired to facilitate swift remediations when system-wide issues are occurring, which is often costly to companies. Given this, we considered our primary challenge to be developing accurate predictions on a binary dataset (anomaly or not anomaly) with highly imbalanced classes. To serve this purpose, we decided to explore the usefulness of a well-known supervised algorithm, Random Forests, which is well-suited for complex prediction problems and is robust against issues like overfitting.

3.3.1 Random Forests

Random Forests operates as a type of ensemble model, using tree-like structure decision making techniques and is trained through bagging or bootstrap aggregating. The outcomes are established based on the predictions made from the decision trees. Since this is a classification problem, the single decision tree is build based on different random samples which are extracted from the population. The decision about the classification class is drawn based on the votes, which is the occurrence of an event in a particular block. The class which gets majority of the votes for classification is then predicted as the output. As the tree grows, the outcomes are returned from the algorithm become increasingly more precise. Ultimately, a random forest becomes a collection of decision trees, where the results of all the individual decision trees are aggregated into a single final result. This allows Random Forests to achieve lower bias and variance than a standard classification tree, and results in this model's biggest advantage; it avoids over-fitting to the data and offers a high precision for predictability. However, the trade-off is higher model complexity, and less interpretability and inference.

Diving a bit deeper, the random forest algorithm follows a sequence of steps with three major components – the topmost is the root node, then is the leaf node and last is the decision node. The branches represent the training dataset divisions. The tree continues to grow until the leaf node is obtained. To help illustrate how this works, please review Figure 5 below [9].

Figure 5:


Tree 1
Tree 2
Tree n
Class 1
Class 1
Class n
Majority Voting for Classification or Averaging for Regression
Final Class

There are three hyperparameters that Random Forests uses to guide its decision-making process:

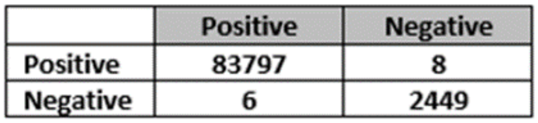
- n-tree: number of trees the algorithm builds before averaging the predictions.
- max_features: maximum number of features random forest considers splitting a node.
- mini_sample_leaf determines the minimum number of leaves required to split an internal node

For our model, we use the default hyperparameters to start, then would assess if any changes or tuning would be required.

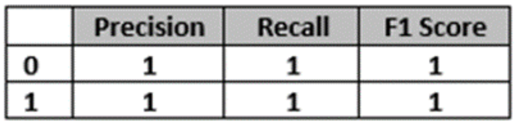
4. Results

The choice of a Random Forests model with this data produced some very accurate predictive results. To obtain these results, we used the log parsers and feature extractors as outlined previously, as well as Python version 3 and scikit learn. Overall, the Random Forests model performed exceedingly well on the test data, and only misclassified fourteen anomalies. This means the model had a classification accuracy of 99.98%. This model also achieved essentially perfect Precision, Recall and F1 scores due to the exceedingly small number of misclassifications compared with number of anomalies in the data, which speaks to the very level of accuracy this model is achieving. These results are summarized in the two tables presented below in Figure 6.

Figure 6:



	Positive	Negative
Positive	83797	8
Negative	6	2449



	Precision	Recall	F1 Score
0	1	1	1
1	1	1	1

5. Discussion

Overall, the Random Forests model performed incredibly well with its out-of-the-box default parameters on this data set. This speaks to its strength as a predictive model in machine learning. However, its predictive power comes at the expense of a lack of interpretability of the model. While detecting anomalies quickly and accurately is a primary concern of industry, having more understanding and insight into how and why anomalies are occurring is also important. This could help inform system enhancements that may prevent the anomalies from occurring in the first place, rather than just remedying the situation when a problem occurs. On this front, we decided to explore the feature importance ranking that resulting from building the Random Forests model. The top fifteen predictors (of the forty-eight given to the model) are summarized in Figure 7 below. These predictors were then matched back to the Event Template that corresponds to the predictor value and is summarized by Figure 8.

Figure 7:

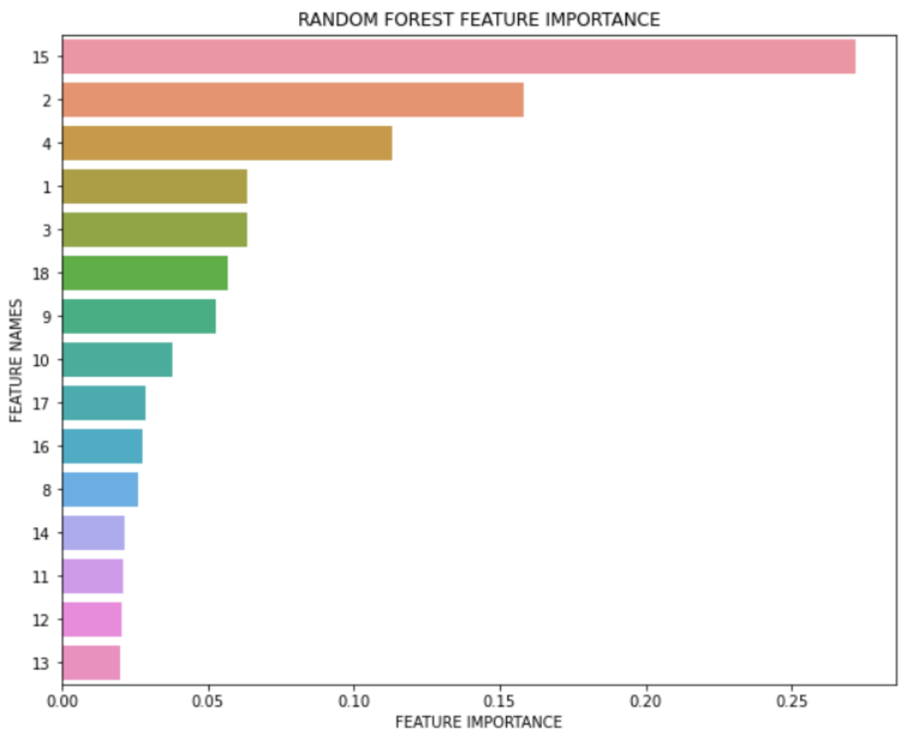


Figure 8:

Predictor	Event Template	Occurrences
15	BLOCK* NameSystem.addStoredBlock: Redundant addStoredBlock request received for <*> on <*> size <*>	975
2	PacketResponder <*> for block <*> <*>	1706728
4	BLOCK* NameSystem.addStoredBlock: blockMap updated: <*> is added to <*> size <*>	1719741
1	BLOCK* NameSystem.allocateBlock: <*> <*>	575061
3	Received block <*> of size <*> from <*>	1706514
18	<*> Starting thread to transfer block <*> to <*>, <*>	165
9	Verification succeeded for <*>	120036
10	<*> Starting thread to transfer block <*> to <*>	6837
17	writeBlock <*> received exception java.io.IOException: Could not read from stream	3226
16	Receiving empty packet for block <*>	1464
8	<*> Served block <*> to <*>	428726
14	Unexpected error trying to delete block <*>. BlockInfo not found in volumeMap.	5545
11	BLOCK* ask <*> to replicate <*> to datanode(s) <*>	6837
12	<*>Transmitted block <*> to <*>	6937
13	Received block <*> src: <*> dest: <*> of size <*>	7097

While this provides some interesting information about what sorts of log messages are contributing to the model predicting an anomaly, there is no obvious pattern to these results. This is a clear limitation of adopting a model which is so heavily focused on prediction accuracy. Future work in this area could be to explore models which support more inferential conclusions to better understand how anomalies are occurring in the system in the first place. Additionally, the pipeline implemented in this project is limited towards certain kind of methodology, which can also be expanded upon in future work.

6. Conclusions

Log files are an important data resource in industry, and one of its many uses can be to help identify anomalies in large scaled distributed systems. The world is moving towards enhancements that reduce manual efforts by automating data processing and analysis efforts. However, there remain some limitations in that log processing is still seen or understood as a developing field. Often, there is not much of awareness in terms of the state-of-the art anomaly detection methods that would provide developers with easily implemented solutions. Instead, developers are left to design or redesign anomaly detection methods from scratch, which is ineffective on many levels. It is our hope that this project and paper shows both how powerful and easy it is to undertake and implement some of these open-source toolkits to achieve highly accurate anomaly prediction results in practice.

References:

[1] Steven Locke, Heng Li, Tse-Hsun (Peter) Chen, Weiyi Shang and Wei Liu, *LogAssist: Assisting Log Analysis Through Log Summarization*. Available: https://users.encs.concordia.ca/~shang/pubs/TSE2021_LogAssist.pdf

[2] Jieming Zhu, Shilin He, Jinyang Liu, Pinjia He, Qi Xie, Zibin Zheng, Michael R. Lyu. Tools and Benchmarks for Automated Log Parsing. *International Conference on Software Engineering (ICSE)*, 2019.

[3] Shilin He, Jieming Zhu, Pinjia He, Michael R. Lyu. Loghub: A Large Collection of System Log Datasets towards Automated Log Analytics. *Arxiv*, 2020.

[4] Shilin He, Jieming Zhu, Pinjia He, Michael R. Lyu. Experience Report: System Log Analysis for Anomaly Detection. IEEE International Symposium on Software Reliability Engineering (ISSRE), 2016.

[5] H. Mi, H. Wang, Y. Zhou, R. Lyu, and H. Cai. Toward fine-grained, unsupervised, scalable performance diagnosis for production cloud computing systems. IEEE Transactions on Parallel and Distributed Systems, 24:1245–1255, 2013.

[6] Pinjia He, Jieming Zhu, Zibin Zheng, Michael R. Lyu. Drain: An Online Log Parsing Approach with Fixed Depth Tree. IEEE International Conference on Web Services (ICWS), 2017.

[7] Pinjia He, Jieming Zhu, Shilin He, Jian Li, Michael R. Lyu. An Evaluation Study on Log Parsing and Its Use in Log Mining. IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2016.

[8] Adetokunbo Makanju, A. Nur Zincir-Heywood, Evangelos E. Milios. Clustering Event Logs Using Iterative Partitioning. ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD), 2009.

[9] E.R Sruthi, *Understanding Random Forest,* Analytics Vidhya, June 2021. Accessed on April 18, 2022. [Online]. Available: https://www.analyticsvidhya.com/blog/2021/06/understanding-random-forest/