Supervised Log File Anomaly Detection with Random Forests and Gradient Boosting

Data 589 - Advanced Machine Learning - Winter 2022 – UBC Okanagan

Sara Hall and Justine Filion

Abstract

Given the vast amounts of data we have access to in the modern world, large-scale data centers have become prevalent and generate massive and complicated system log files. These are difficult for humans to use for detecting when something has gone wrong. As a result, machine learning is uniquely positioned to fill this gap and help perform automatic anomaly detection in log files. With this problem in mind, we chose a dataset generated from the Hadoop Distributed File System (HDFS) with messages split into blocks and labelled with either "Anomaly" or "Normal". To parse the file, we used the Drain method which split the log messages into forty-eight different event types. We then made an event count matrix and used the frequency of the event types as predictors for whether an anomaly had occurred in a block. Because we were focused on predicting anomalies as well as possible, we tuned both a gradient boosting model and a random forest on 80% of the data using 5-fold cross-validation. Both models achieved high F1 scores on the test data, indicating that both performed well. However, the random forest was superior and achieved a recall score of 1 on the test data, indicating that it classified all anomalies correctly.

1. Introduction

We are currently living in the age of data. It is estimated that ninety percent of the data we have today has been generated over the past two years [1]. That means we have access to astronomical and unprecedented amounts of data, but individual local computers do not have the capacity to store or process it. As a result, the past few years have seen a rise in large-scale distributed computing systems in which data can be stored and processed across many different computing clusters, each made up of several computers. Apache Hadoop provides a framework for distributed data processing, and one of the core concepts is the distributed file system (HDFS), which allows for the distributed storage of substantial amounts of data [2]. One of the hurdles with the distributed approach is that the complexity makes it harder to diagnose problems when they occur. Lots of software make use of logging and print statements to alert the user to what is going on in the background and historically developers have often used this information to diagnose bugs by manually reading or using targeted keyword searches [3], [4]. However, in the case of large-scale distributed file systems, there often ends up being too much information for a human to quickly detect when something goes wrong. As a result, machine learning is uniquely positioned to fill this gap and rapidly predict anomalies using the information that is present in the semi-structured log files. Of course, to do this, the log files need to be analyzed in some way to pull out information that can be fed into a machine-learning algorithm and then that algorithm must be tuned to perform as well as possible, which is not an easy task [4]. As a result, in this paper, we make use of a dataset consisting of some log outputs from HDFS which are labelled as either '*Anomaly*' or '*Normal'*. Due to the presence of labels, we focused on developing a pipeline that parses the output log files, extracts features in the form of an event count matrix, and then compares various supervised learning methods for anomaly prediction. By doing so, we hope to contribute to the growing body of literature that uses machine learning techniques to facilitate anomaly detection using the massive log files output by large-scale distributed computing systems.

2. Literature Review

There is a large body of literature focused on both automatic log file parsing and anomaly detection. However, for the purpose of this project, we will focus on a cursory review of the most basic and relevant information.

2.1 Log File Structures

Logs are generally considered to be free-form unstructured text; however, they are typically generated by print statements in source code. This means that at the end of the day, there are a finite number of message types and there is some structure to them [3]. Previous work has often split the messages into two portions – variable and constant. The constant portion refers to what would effectively be the constant portions of strings that are printed, while the variable portion refers to what changes when the print statement is executed [3]–[6]. For instance, if the source file had something like print("Server", server_id, "received", packet_size, "bytes"), then the constant part of the message would be "Server received bytes", while the variable parts would be the server ID and packet size.

2.1 Approaches to Log Parsing

Typically, the constant parts of the messages are what are more useful for anomaly detection, so methods for log parsing usually focus on pulling out that information. There are two general methods for log parsing - clustering and heuristic [7]. Clustering methods involve calculating the distance between messages and then clustering them into event groups based on which ones are closest together. In contrast, heuristic methods like Drain [5] use the frequency and position of words in the message to classify them into event types. Regular expressions can also sometimes be used to increase the accuracy of this process [5]. Regardless of which method type is used, you generally end up with a data frame that has at least a timestamp and event type. In the case of Drain, you also end up with the constant and variable portions of the messages included in the output data frame [5].

2.2 Anomaly Detection Methodology

By classifying the log file messages into different event types, the frequency of the event on their own or in combination with their sequence can be used to generate what are referred to as features which are fed into machine learning algorithms for anomaly prediction [3], [4], [7]. Because anomalies in log files are usually unlabelled unless someone has gone through and done it by hand, a lot of prior work has focussed on using unsupervised methods for anomaly detection in log files [4], [6]. However, the HDFS file that we used in our work was hand-labelled, so we chose to focus on supervised methods for binary classification. Previous work has made use of decision trees, random forests, logistic regression, and supervised neural networks, among other approaches for this task [3], [4], [6], [7].

3. Background

Based on the previous modelling approaches, we implemented a random forest and gradient boosting. Our goal was to predict anomalies as accurately as possible rather than to understand what events lead to the occurrence of an anomaly, so we chose models that tend more to the "black-box" side of things. These models are generally good at making accurate predictions but tend to be harder to use for explaining the relationships between the predictor variables and the response variable [8].

3.1 Random Forests

Random forest classification is a great supervised machine learning method built on top of decision trees [9]. The idea behind decision trees is to come up with binary splits based on the predictor variables which minimize the classification error when predicting the response variable. To avoid overfitting, a large tree is grown with lots of these splits and then pruned back using a penalty based on the number of terminal nodes in the tree combined with a tuning parameter, α. Decision trees are great because they have variable selection built-in. If a predictor is not useful, then none of the splits in the tree will be based on it. By their hierarchical nature in making decisions, they also inherently model any interactions between the predictors. However, they tend to be an oversimplification, so they have high bias and don't fit the training data very well and have high variance because they also tend to not perform well on new data [10].

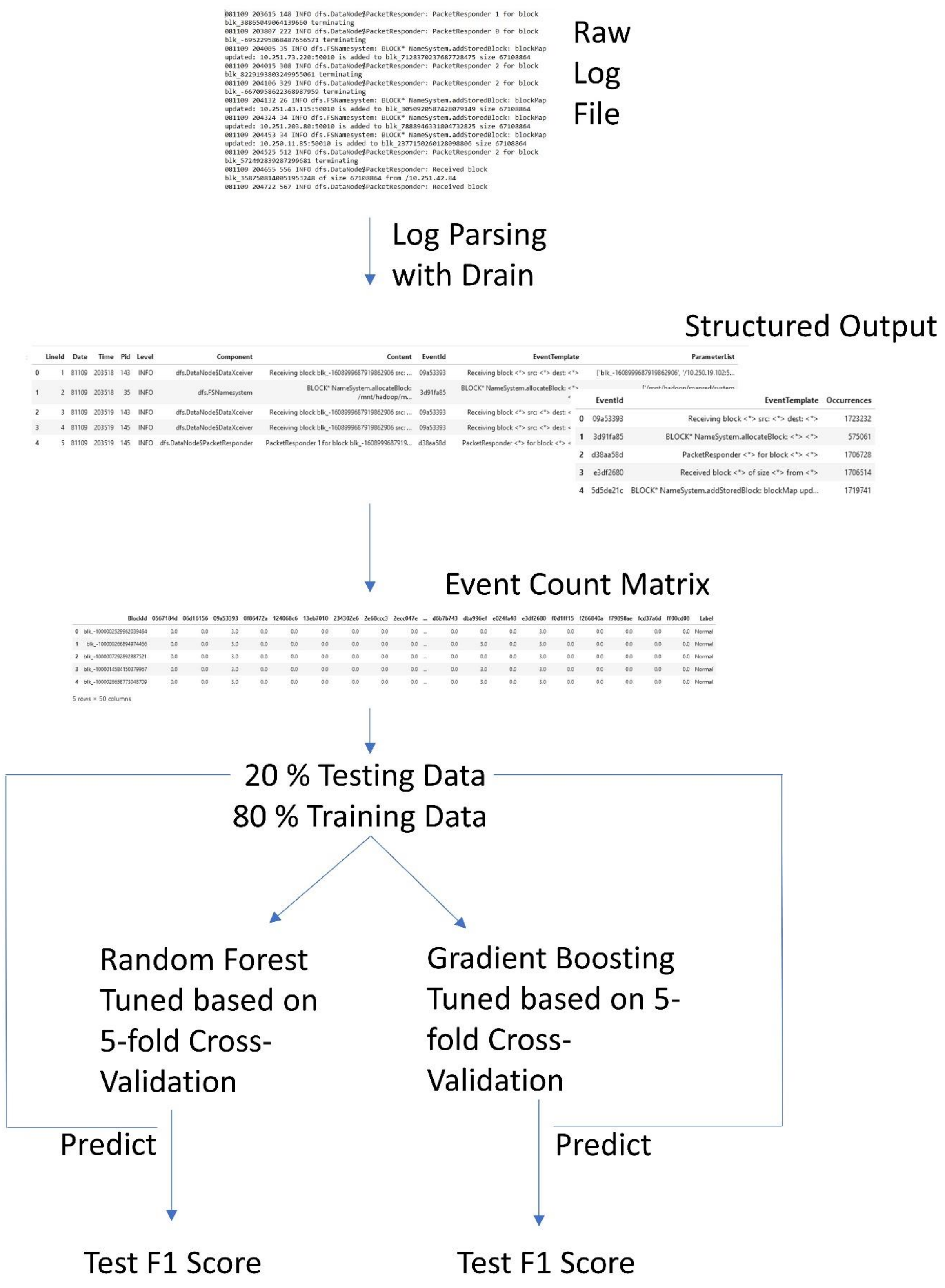
Random forests address these issues with decision trees by creating lots of trees from bootstrap samples, randomly removing a certain number of predictors from consideration at each split, and then averaging together the results from all of the trees. By averaging the results from trees made from bootstrap samples, the results generalize much better to new data. Additionally, only considering a certain number of parameters at each split means that the generated trees are less correlated with each other, further improving predictive results on new data [9], [11]. This averaging greatly improves the predictive capability of the model but does mean that it's much harder to explain how the predictor variables influence the response.

3.2 Gradient Boosting

Similar to random forests, gradient boosting is a model that improves the performance of the simple decision tree. However, the way it does so is quite different. Gradient boosting works by generating a sequence of trees constrained to a small depth such that each new tree in addition to the previous ones results in a minimization of the loss function. This iterative process of fitting trees and adding their results together scaled with a learning parameter is often referred to as learning slowly and has been shown in the past to greatly improve the predictive performance of decision trees [12].

4. Methodology

In this section, we will discuss the various steps that were undertaken to predict whether a data block from an HDFS log file contained an anomaly. An overview of what was performed is outlined in Figure 1.



```
081109 203615 148 INFO dfs.DataNode$PacketResponder: PacketResponder 1 for block
blk_38865049064139660 terminating
081109 203807 222 INFO dfs.DataNode$PacketResponder: PacketResponder 0 for block
blk_-6952295868487656571 terminating
081109 204005 35 INFO dfs.FSNamesystem: BLOCK* NameSystem.addStoredBlock: blockMap
updated: 10.251.73.220:50010 is added to blk_7128370237687728475 size 67108864
081109 204015 308 INFO dfs.DataNode$PacketResponder: PacketResponder 2 for block
blk_8229193803249955061 terminating
081109 204106 329 INFO dfs.DataNode$PacketResponder: PacketResponder 2 for block
blk_-6670958622368987959 terminating
081109 204132 26 INFO dfs.FSNamesystem: BLOCK* NameSystem.addStoredBlock: blockMap
updated: 10.251.43.115:50010 is added to blk_3050920587428079149 size 67108864
081109 204324 34 INFO dfs.FSNamesystem: BLOCK* NameSystem.addStoredBlock: blockMap
updated: 10.251.203.80:50010 is added to blk_7888946331804732825 size 67108864
081109 204453 34 INFO dfs.FSNamesystem: BLOCK* NameSystem.addStoredBlock: blockMap
updated: 10.250.11.85:50010 is added to blk_2377150260128098806 size 67108864
081109 204525 512 INFO dfs.DataNode$PacketResponder: PacketResponder 2 for block
blk_572492839287299681 terminating
081109 204655 556 INFO dfs.DataNode$PacketResponder: Received block
blk_3587508140051953248 of size 67108864 from /10.251.42.84
081109 204722 567 INFO dfs.DataNode$PacketResponder: Received block
```

	LineId	Date	Time	Pid	Level	Component	Content	EventId	EventTemplate	ParameterList
0	1	81109	203518	143	INFO	dfs.DataNode$DataXceiver	Receiving block blk_-1608999687919862906 src: ...	09a53393	Receiving block <*> src: <*> dest: <*>	['blk_-1608999687919862906', '/10.250.19.102:5...
1	2	81109	203518	35	INFO	dfs.FSNamesystem	BLOCK* NameSystem.allocateBlock: /mnt/hadoop/m...	3d91fa85	BLOCK* NameSystem.allocateBlock: <*> <	['/mnt/hadoop/mapred/system
2	3	81109	203519	143	INFO	dfs.DataNode$DataXceiver	Receiving block blk_-1608999687919862906 src: ...	09a53393	Receiving block <*> src: <*> dest: <	
3	4	81109	203519	145	INFO	dfs.DataNode$DataXceiver	Receiving block blk_-1608999687919862906 src: ...	09a53393	Receiving block <*> src: <*> dest: <	
4	5	81109	203519	145	INFO	dfs.DataNode$PacketResponder	PacketResponder 1 for block blk_-1608999687919...	d38aa58d	PacketResponder <*> for block <*> <	

	EventId	EventTemplate	Occurrences
0	09a53393	Receiving block <*> src: <*> dest: <*>	1723232
1	3d91fa85	BLOCK* NameSystem.allocateBlock: <*> <*>	575061
2	d38aa58d	PacketResponder <*> for block <*> <*>	1706728
3	e3df2680	Received block <*> of size <*> from <*>	1706514
4	5d5de21c	BLOCK* NameSystem.addStoredBlock: blockMap upd...	1719741

	BlockId	0567184d	06d16156	09a53393	0f86472a	124068c6	13eb7010	234302e6	2e68ccc3	2ecc047e	...	d6b7b743	dba996ef	e024fa48	e3df2680	f0d1ff15	f266840a	f79898ae	fcd37a6d	ff00cd08	Label
0	blk_-1000002529962039464	0.0	0.0	3.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	3.0	0.0	0.0	0.0	0.0	0.0	Normal
1	blk_-100000266894974466	0.0	0.0	3.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	3.0	0.0	3.0	0.0	0.0	0.0	0.0	0.0	Normal
2	blk_-1000007292892887521	0.0	0.0	3.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	3.0	0.0	0.0	0.0	0.0	0.0	Normal
3	blk_-1000014584150379967	0.0	0.0	3.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	3.0	0.0	3.0	0.0	0.0	0.0	0.0	0.0	Normal
4	blk_-1000028658773048709	0.0	0.0	3.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	3.0	0.0	3.0	0.0	0.0	0.0	0.0	0.0	Normal

5 rows × 50 columns

Figure 1. A general outline of the methodology performed and further explained in this report.

4.1 Automated Log Parsing: Drain

The initial step consisted of transforming the unstructured data from the log files into a format that could eventually be fed to a machine-learning algorithm. To do this, we made use of an automated log parsing method called Drain, which is a fixed depth tree-based online log parsing method [5]. As its name suggests, Drain adopts a parser with a tree structure of specified depth to extract the various event templates from the log messages. Drain then calculates the similarity between the log messages and each log event. All the messages are then matched with the event that is most similar to it, effectively classifying the messages into a concise number of events.

We can briefly expand on how this parser makes use of a tree structure to determine the list of event templates. Each tree node describes rules for how to search the log messages [5]. First, the messages are split according to the number of tokens (i.e., words, groups of characters) that they contain. For example, the message "Received block 23i1og" contains 3 tokens. This is performed because it is assumed that messages that belong to the same event are more likely to be of the same length. Next, the messages are classified according to their first token as we observe that they are often constant (i.e., Received, Send, Starting). If the depth of the tree is fixed to 3, then these are the only 2 rules that are considered when extracting the log events (see Figure 2 for a simplified tree parser of depth 3). If the fixed depth is set to 4, then both the first and second tokens of each message are considered. The number of tokens considered corresponds to the depth of the tree minus two. Finally, it is worth mentioning that regular expressions can be passed to Drain to pre-process the raw log files in the hopes of improving the parsing accuracy. In our case, regular expressions were used to remove the Block ID and the IP addresses from the messages to facilitate the log parsing process.

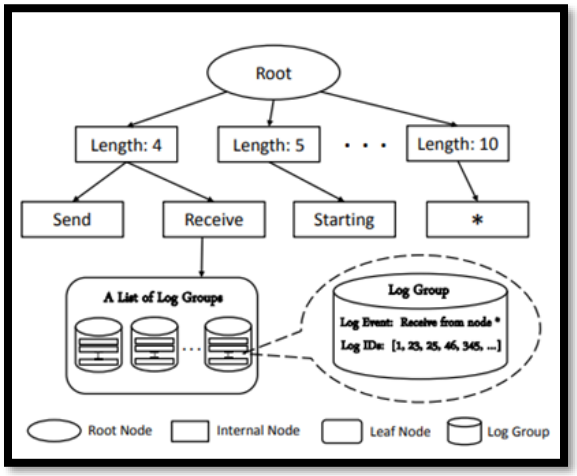


Figure 2. Parser tree of depth 3 (taken from the original Drain paper [5])

4.2 Feature Extraction: Event Count Matrix

After the parsing of the log files, every log message had been attributed an event ID corresponding to the event that was most similar to it. Furthermore, as per the structure of the log files, each message was also associated with the block ID that it belonged to. As was mentioned previously, it is the blocks that were labelled either 'Anomaly' or 'Normal'. Thus, it became essential to summarize the event information in a way that related back to the blocks. Keeping this in mind, we decided to make an event count matrix X where element $X_{i,j}$ represents how many times the event j occurred in the i -th block [7]. From the parser, we retrieved 48 distinct events and because we had 575 061 different block ID's, the dimension of this matrix was $575\ 061\ X\ 48$.

We used this matrix as the input to the different machine-learning models we implemented. Being in the context of supervised learning, we also had a response variable which corresponded to the label of each of these 575 061 blocks. If a block was labelled *Anomaly*, the corresponding response variable took the value 1 whereas if the block was labelled *Normal*, it took the value 0.

4.3 Modeling

With the event count matrix, we were now ready to implement machine-learning models. As was previously mentioned, we focused on training two different types of models: random forests and extreme gradient boosting models (XGBoost).

For us to be able to evaluate the models accordingly, we needed to split our data into 2 separate sets, one used to train our models and the other to test their performance on observations they had never seen before. We implemented an 80/20 train/test split. Furthermore, when tuning the hyperparameters for the random forest, we performed a grid search with 3-fold cross-validation and selected the combination of hyperparameters that resulted in the highest F1 score.

We chose to use the F1 score because it is composed of two other metrics: precision and recall, both of which are well suited for unbalanced classes [13]. Our objective was to correctly identify as many of the anomalies as possible, the true positives. Consequently, we needed to choose metrics that did not account for the true negatives, which in our case represented the correctly identified *'Normal'* cases.

A metric such as accuracy, which measures the proportion of predictions our model correctly identifies, would be hugely deceiving. For example, because our dataset consisted of only 2.93% anomalies, obtaining an accuracy of approximately 97% could simply mean that our model identified all blocks as being *'Normal'*, leaving all the anomalies misclassified. In our context, precision measures the number of anomaly predictions that were true anomalies and recall measures the proportion of anomalies that we were correctly able to predict.

For the extreme gradient boosting model, implementing a grid search proved to be computationally expensive. We thus decided to implement Bayesian optimization, once again using the F1 score as our tuning metric. The advantage of using Bayesian optimization is that it gains insight from the performance of the past set of hyperparameters and uses it to redefine the search space for the next iteration. In doing so, the computational efficiency is increased [14].

Having successfully tuned our models, we were then able to evaluate their performance using various cross-validated metrics such as balanced accuracy, precision, recall, F1 score and log loss. We used 5-fold cross-validation. Based on these cross-validated results, we chose to compete the best performing random forest model against the best performing boosting model on the testing dataset.

The scikit-learn machine learning library in Python was used to train all the models as well as for evaluating them [15].

5. Results

5.1 Hyperparameters

We fit 2 different random forest models. The first one was obtained using default hyperparameters. Next, we tuned these hyperparameters and obtained the values shown in Table 1.

Table 1: Tuning hyperparameters for random forest model.

Hyperparameter	Value
Number of trees in the forest (n_estimators)	100
Weights associated with the classes (class_weight)	None
Number of features considered at each split (max_features)	log2
Minimum number of samples to split a node (min_samples_split)	20

A value of '*None'* for the class_weight parameter signifies that the classes will all be attributed a weight of 1, thus equal weighting for all classes. A value of '*log2'* for the max_features parameter signifies that the number of features considered at each split is $\log_2(number\ of\ features)$ [16]. Having 48 predictors, this corresponds to approximately 6.

Similarly, we also fit 2 different XGBoost models; one with the default hyperparameters and the other with tuned hyperparameters. The tuned hyperparameters are shown in Table 2.

Table 2: Tuning hyperparameters for extreme gradient boosting model.

Hyperparameter	Value
Factor by which the contribution of each tree is shrunk (learning_rate)	0.4203
Maximum depth of the individual trees (max_depth)	3
Number of boosting stages (n_estimators)	266
Fraction of samples used for fitting the individual trees (subsample)	0.8948

Having a value for the 'subsample' parameter that is smaller than 1 signifies that our boosting algorithm uses stochastic gradient descent to optimize the loss function. It also leads to a reduction in variance but simultaneously increases the bias [17].

5.2 Variable Importance

Variable importance allows us to assess which features were the most useful at predicting the response variable. In our case, this corresponds to assessing which event was most important in predicting the label of each block.

We used the mean decrease in impurity (MDI) to evaluate said variable importance. This corresponds to the "total decrease in node impurity averaged over all the trees of the ensemble"

[18]. Thus, a feature having a high importance means that it often led to splits that resulted in its nodes being more homogeneous. Figure 3 shows the 10 most important variables for both models.

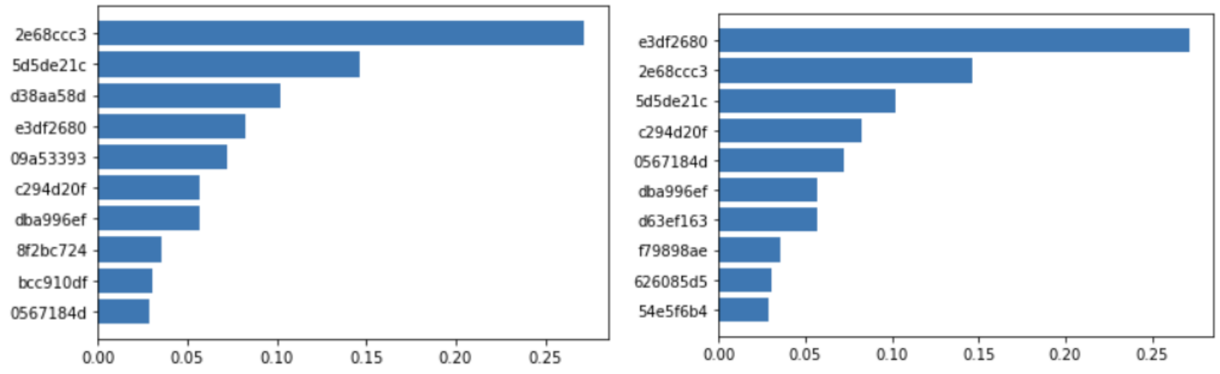


Figure 3. Variable Importance for Random Forest (left) and XGBoost (right).

We notice that 3 events are among both the Random Forest and the XGBoost models' top 5 most important features. These events are summarized in table 3 below.

Table 3: Event IDs that are in both models' top 5 most important variables.

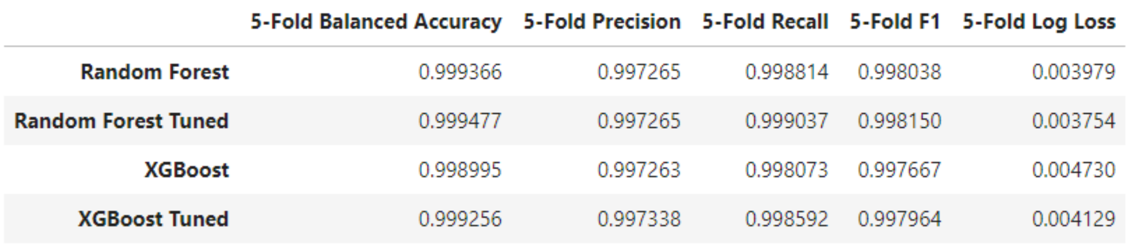
Event ID	Event Template
2e68ccc3	Unexpected error trying to delete block <*>. BlockInfo not found in volumeMap.
e3df2680	Received block <*> of size <*> from <*>
5d5de21c	BLOCK*NameSystem.addStoredBlock:blockMap updated: <*> is added to <*> size <*>

It is interesting to see that event 2e68ccc3, the most important feature for the Random Forest model and the second most important for the XGBoost model is in fact an error message. It is thus unsurprising that it was very helpful in deciphering whether or not a block contained an anomaly. Greater domain expertise in log files would help provide plausible causes for these errors.

5.3 Cross-validated Metrics

In table 4 below we see that unsurprisingly, in the case of the Random Forest and the XGBoost, the models that are tuned performed better than the ones with the default hyperparameters. Furthermore, when comparing the two tuned models to one another, we notice that the Random Forest outperforms the XGBoost in all metrics except precision. Having said that, both models perform outstandingly well with balanced accuracy, precision, recall and F1 scores all above 0.997 and with log losses very close to 0.

Table 4: 5-Fold Cross-validated Metrics.

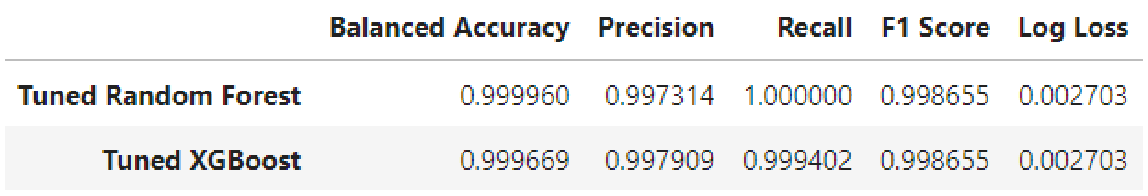


	5-Fold Balanced Accuracy	5-Fold Precision	5-Fold Recall	5-Fold F1	5-Fold Log Loss
Random Forest	0.999366	0.997265	0.998814	0.998038	0.003979
Random Forest Tuned	0.999477	0.997265	0.999037	0.998150	0.003754
XGBoost	0.998995	0.997263	0.998073	0.997667	0.004730
XGBoost Tuned	0.999256	0.997338	0.998592	0.997964	0.004129

5.3 Performance on Testing Data

To obtain an unbiased idea of how these models could perform on data they have never seen before, we measured their performance on the testing data which represented 20% of the initial dataset. This corresponded to 460 048 observations. The results obtained are illustrated in table 5 below.

Table 5: Metrics Obtained on Testing Data.



	Balanced Accuracy	Precision	Recall	F1 Score	Log Loss
Tuned Random Forest	0.999960	0.997314	1.000000	0.998655	0.002703
Tuned XGBoost	0.999669	0.997909	0.999402	0.998655	0.002703

These metrics are similar to those obtained using 5-fold cross-validation and surprisingly even slightly better. This might be caused by our test set containing easier cases to predict. Once again, the Random Forest performs slightly better than the XGBoost model in terms of balanced accuracy and recall whereas boosting still achieves a higher precision. The other metrics are equivalent for both models.

5.4 Model Selection

Although the models both performed extremely well and could be considered for future log anomaly detection, we consider our tuned Random Forest model to be better. Not only did it allow us to obtain better results for most of the selected metrics, but it was also less computationally expensive to tune and fit.

References

[1] "25+ Impressive Big Data Statistics for 2022." https://techjury.net/blog/big-data-statistics/ (accessed Apr. 19, 2022).

[2] "Apache Hadoop." https://hadoop.apache.org/ (accessed Apr. 19, 2022).

[3] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting Large-Scale System Problems by Mining Console Logs".

[4] Z. Chen *et al.*, "Experience Report: Deep Learning-based System Log Analysis for Anomaly Detection; Experience Report: Deep Learning-based System Log Analysis for Anomaly Detection," 2021, doi: 10.1145/1122445.1122456.

[5] P. He, J. Zhu, Z. Zheng, and M. R. Lyu, "Drain: An Online Log Parsing Approach with Fixed Depth Tree," *Proceedings - 2017 IEEE 24th International Conference on Web Services, ICWS 2017*, pp. 33–40, Sep. 2017, doi: 10.1109/ICWS.2017.13.

[6] S. Lu, X. Wei, Y. Li, and L. Wang, "Detecting Anomaly in Big Data System Logs Using Convolutional Neural Network; Detecting Anomaly in Big Data System Logs Using Convolutional Neural Network," 2018, doi: 10.1109/DASC/PiCom/DataCom/CyberSciTec.2018.00037.

[7] S. He, J. Zhu, P. He, and M. R. Lyu, "Experience Report: System Log Analysis for Anomaly Detection," *Proceedings - International Symposium on Software Reliability Engineering, ISSRE*, pp. 207–218, Dec. 2016, doi: 10.1109/ISSRE.2016.21.

[8] "Inference vs Prediction - Data Science Blog: Understand. Implement. Succed." https://www.datascienceblog.net/post/commentary/inference-vs-prediction/ (accessed Apr. 21, 2022).

[9] L. Breiman, "Random Forests," *Machine Learning 2001 45:1*, vol. 45, no. 1, pp. 5–32, Oct. 2001, doi: 10.1023/A:1010933404324.

[10] "data571/lecture3.pdf at main · ubco-mds-2021/data571." https://github.com/ubco-mds-2021/data571/blob/main/lectures/lecture3.pdf (accessed Apr. 21, 2022).

[11] "data571/lecture4.pdf at main · ubco-mds-2021/data571." https://github.com/ubco-mds-2021/data571/blob/main/lectures/lecture4.pdf (accessed Apr. 21, 2022).

[12] "A Gentle Introduction to the Gradient Boosting Algorithm for Machine Learning." https://machinelearningmastery.com/gentle-introduction-gradient-boosting-algorithm-machine-learning/ (accessed Apr. 21, 2022).

[13] "The F1 score | Towards Data Science." https://towardsdatascience.com/the-f1-score-bec2bbc38aa6 (accessed Apr. 21, 2022).

[14] V. H. Nguyen *et al.*, "Applying Bayesian Optimization for Machine Learning Models in Predicting the Surface Roughness in Single-Point Diamond Turning Polycarbonate," *Mathematical Problems in Engineering*, vol. 2021, 2021, doi: 10.1155/2021/6815802.

[15] “scikit-learn: machine learning in Python — scikit-learn 1.0.2 documentation.” https://scikit-learn.org/stable/ (accessed Apr. 21, 2022).

[16] “sklearn.ensemble.RandomForestClassifier — scikit-learn 1.0.2 documentation.” https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html (accessed Apr. 21, 2022).

[17] “sklearn.ensemble.GradientBoostingClassifier — scikit-learn 1.0.2 documentation.” https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html (accessed Apr. 21, 2022).

[18] “Feature Importance Measures for Tree Models — Part I | by Ceshine Lee | Veritable | Medium.” https://medium.com/the-artificial-impostor/feature-importance-measures-for-tree-models-part-i-47f187c1a2c3 (accessed Apr. 22, 2022).