ML APPLICATIONS ON OPENSTACK LOG DATA ANALYSIS

JUNE - AUG 2019

AUTHOR(S):
Ravi Charan Nudurupati
Openlab Summer Intern - 2019
ravicharan.vsp@gmail.com

SUPERVISOR(S):
Domenico Giordano,
CERN

CERN openlab

PROJECT SPECIFICATION

The numerous logs produced daily are a major source of error reporting in Openstack. On average, 100k logs are generated every hour generating nearly 3TB's of data per day. Thus, it would be immensely difficult to manually check the historical logs to find similar messages. This project aims at building mathematical models to detect anomalies in the error logs produced by the Openstack server and filter them out in real-time. A mixture of algorithms like MinHash, Locality Sensitive Hashing, and DB Scan have been used to create the necessary model. The Rally log data generate through the course of a month has been used to train our model. Further details are provided in the coming sections. The primary work has been done on Python and the notebooks shall always remain available on a Github repository.

ABSTRACT

A massive amount of data is generated by the Openstack cloud services in the format of service logs. Besides timestamps and log level fields, these logs contain additional information useful for pattern analysis. Unfortunately, this information is generally exposed in semi-structured text format, not allowing direct analysis without additional munging of the data. Traditional approaches to extract information from those fields are rule-based, mainly applying regular expressions upon knowledge of the text structure. These approaches require a pre-knowledge of all text patterns and are not scalable with the growth of the service. We propose a solution that is a mixture of the MinHash Locality Sensitive Hashing and the DB scan algorithm for data clustering.

TABLE OF CONTENTS

1. INTRODUCTION

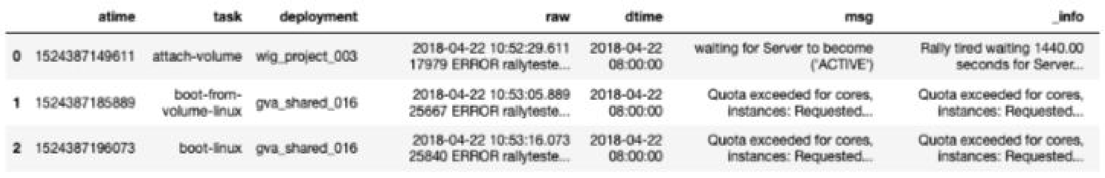
In the operation of a data-center, one of the major challenges is to be able to categorize errors automatically in order to take curative action for monitoring purposes. The data centers are managed by deploying services and all of these services deployed produce logs mainly in a semi-structured format. The logs are collected for postmortem analysis on a log level.

Analyzing the log data to identify patterns and problems has been attempted in numerous ways over the past couple of years. At its most general, "to log" is "to put information into a written record" and it is a process as old as writing systems themselves. Previously, a manual investigation was used where the administrators would read the logs stored on each computer, perhaps aided by grepping for keywords. As the scale of services has increased in the age of Big Data, this approach is no longer feasible.

Attempts have been made in the last 20 years to automate this process. One elementary method that is investigated regularly and is a classic example of expert systems that emulate the human expert decision-making process is the use of regular expressions (regex), nested if-then rules used to select key features. However, this requires prior knowledge of the dataset, and is proportional in difficulty to the number of distinct message types present in the data and requires continuous updating if the nature of log file changes.

In this paper, a solution is proposed that is a mixture of the MinHash Locality Sensitive Hashing and the DB scan algorithm for data clustering.

Our primary concern is to filter out the logs that contain the error messages. These messages are analyzed and then clustered into different classes. An example of the log error messages produced could be seen in Figure 1.



	atime	task	deployment	raw	dtime	msg	_info
0	1524387149611	attach-volume	wig_project_003	2018-04-22 10:52:29.611 17979 ERROR rallyteste...	2018-04-22 08:00:00	waiting for Server to become ('ACTIVE')	Rally tired waiting 1440.00 seconds for Server...
1	1524387185889	boot-from-volume-linux	gva_shared_016	2018-04-22 10:53:05.889 25667 ERROR rallyteste...	2018-04-22 08:00:00	Quota exceeded for cores, instances: Requested...	Quota exceeded for cores, instances: Requested...
2	1524387196073	boot-linux	gva_shared_016	2018-04-22 10:53:16.073 25840 ERROR rallyteste...	2018-04-22 08:00:00	Quota exceeded for cores, instances: Requested...	Quota exceeded for cores, instances: Requested...

Figure 1: Error Logs example.

The basic idea is to analyze and cluster the error messages into different classes. One can see from the error logs that a few features in all the messages are unique (eg. time-stamps, ErrorID.. etc). The clustering algorithm runs on the whole dataset and extracts features from every cluster and marks those set of features as the representative feature-set of the cluster. When a new entry is fed into the database, it is compared with the principal features of all the clusters and then it is either placed in a suitable cluster or marked as an anomaly.

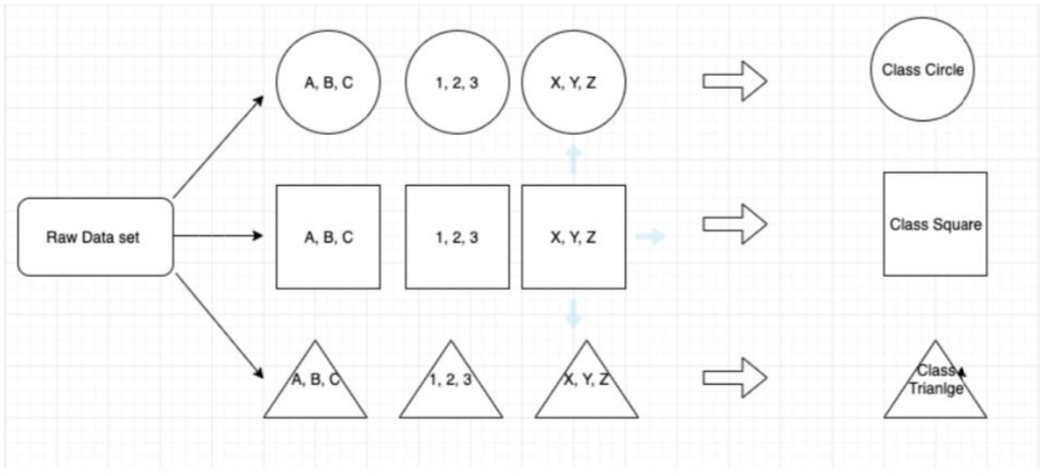


Figure 2: Visual representation of the workflow

Figure 2 shows the visual representation of the algorithm. Each one of the 3 shapes represents a class and the variables inside each shape represent the unique tokens that vary for every message in the dataset. The data is clustered based on the similarity of the shapes ignoring the unique tokens of the message. Once the messages are clustered, the principal features are extracted from each class and marked as a feature vector of that class. Now, every new entry in the data is compared with these principal feature vectors to classify the new entry into one of the classes.

2. DATASET

The numerous logs produced daily are a major source of error reporting in Openstack. On average, 100k logs are generated every hour generating nearly 3TB’s of data per day. Thus, it would be immensely difficult to manually check the historical logs to find similar messages.

The initial phase of the clustering process is grouping the data into common clusters by the number of tokens. A token is either an independently existing character or a word in a log message. Ideally, since the error messages are generated by the machine, the number of tokens would be same for the all the messages of the same class. The histogram below shows the distribution of the frequency of the logs according to the log message length in characters.

3. DATA VISUALIZATION

We use a measure called the "Jaccard similarity" to compute the similarity between the messages. Jaccard similarity is a value that ranges between 0 and 1 in which exactly the same messages have a similarity value of 1 and entirely different ones have a similarity measure of 0. The Jaccard distance is measured as *1 - Jaccard Similarity*. This will be further discussed in the methodology section. Our clustering algorithm relies on a Jaccard similarity threshold which is used as a cut-off to classify two messages to fall under the same cluster.

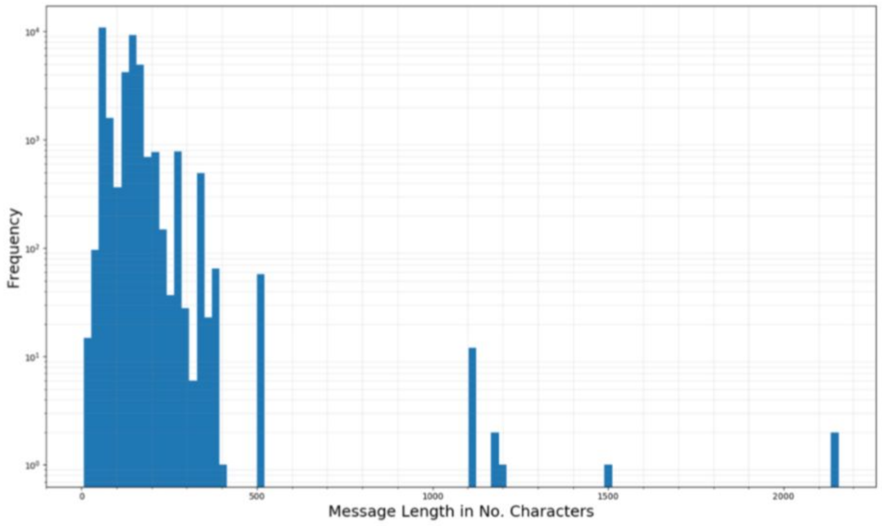


Figure 3: Histogram showing the distribution of number of logs according to the message length

Since the initial phase of clustering is separating the messages by token length, we iterate through all the clusters of a different token length and run our algorithm to divide it into further clusters.

The Jaccard between a small subset of the log strings is reported in Figure 4 as a 2D heat-map between the message pairs. The x-axis of the plot identifies the index of the message and the y-axis indicates the string(or the message itself). The value of the Jaccard distance is reported in a color map, where dark colors show similar messages (distance near 0) and light colors show different messages (distance near 1). The matrix is symmetric because the Jaccard similarity is symmetric J(A, B)=J(B, A). The Jaccard similarity of the identity is J(A, A)=1, this indicates that the messages are essentially the same.

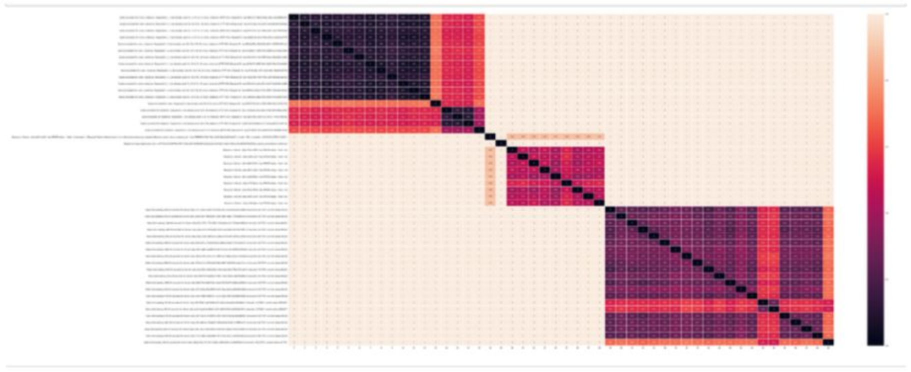


Figure 4: Jaccard Distance heatmap.

In the heat-map in Figure 4, each square shows the Jaccard similarity between each set of message pairs, hence it is represented as a 2D map that shows all the possible combinations.

Figure 5 reports the density distribution of the Jaccard distances for the number of tokens in each message in the form of a Violin-Seaborn plot. As one can see in figure 5 that all the values are distributed between 1 and 0. If a set of messages is distributed at 1, it means that the messages are essentially the same with a 100 percent similarity. On the other hand, the distribution at 0 indicates that the messages are entirely different with no intersecting tokens between them.

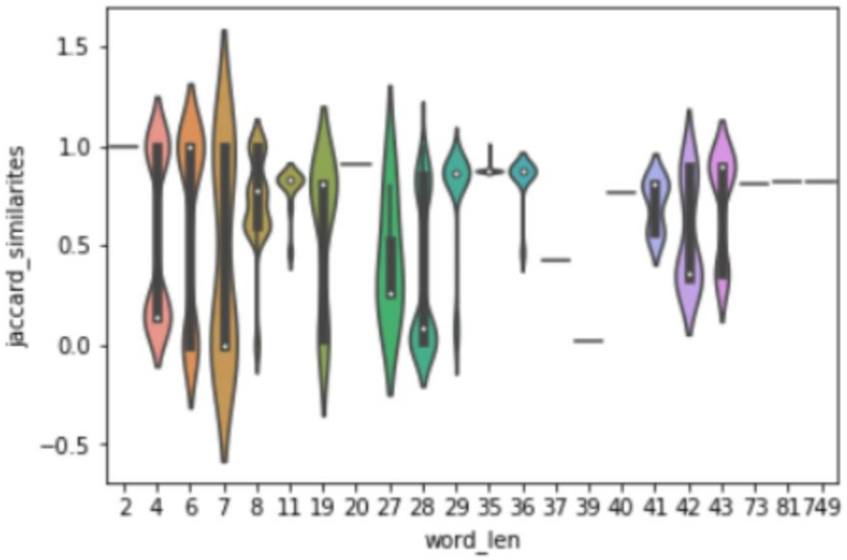
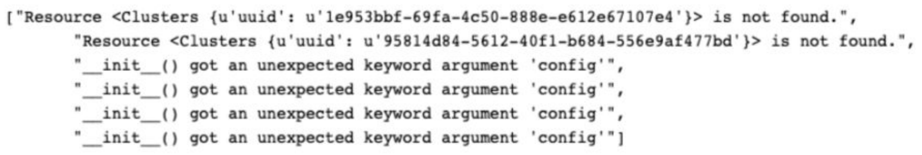


Figure 5: Violin Seaborn distribution

We take the observation on messages with token length = 7 as shown in Figure 6, we observe two classes of messages here. So the Jaccard distance is distributed across [0, .75, 1]. Here the frequency of .75 is one because only two messages are almost similar. Hence, the plot it wider in the regions around (0, 1).



```
["Resource <Clusters {u'uuid': u'1e953bbf-69fa-4c50-888e-e612e67107e4'}> is not found.",
       "Resource <Clusters {u'uuid': u'95814d84-5612-40f1-b684-556e9af477bd'}> is not found.",
       "__init__() got an unexpected keyword argument 'config'",
       "__init__() got an unexpected keyword argument 'config'",
       "__init__() got an unexpected keyword argument 'config'",
       "__init__() got an unexpected keyword argument 'config'"]
```

Figure 6: All the messages in the data-set with token length = 7

Among all the 34k log messages on which algorithm is run[6], we observe that a few entries like 37 and 39 have a single instance for Jaccard in the plot, this is because the messages with the token length 37 and 38 have only 2 entries and hence a 1x1 Jaccard similarity matrix.

4. MÉTHODOLOGIES

The first step is to extract message "keys" which correspond to messages generated by the same lines of code but with different message variables, implemented using regex. They proceed to utilize the similarity in clustering raw log keys by using the edit distance as their metric, which is defined as the number of insertion, deletion, or replacement operations required to convert one message into another. This distance does not account for the position of the words in the logs, and so they use the weighted edit distance which uses a sigmoid function to place greater emphasis on words appearing earlier in the message than later based on the idea/observation that programmers tend to place important information in this way. The edit distances to all other logs are calculated, and all members whose distance is less than a threshold is connected via a link.

This section defines a mathematical approach to the problem of identifying similar messages and clustering them. In order to identify similar items, we must first identify how to define and quantify similarity.

a. MinHash

In order to identify similar items, a similarity measure is necessary. This approach uses a similarity measure called the Jaccard similarity. Jaccard Similarity treats a string (which is a log message in our case) as a set of tokens (words). The similarity between the two messages is defined as the number of similar tokens between the two given messages. Mathematically, Jaccard similarity J between two given set of words A and B is given by

$$J = (A \wedge B) / (A \vee B)$$

Other alternative methods include measuring distance as the number of edits required to transform one message in another. The signatures that are desired to construct for sets are composed of the results of a large number of calculations, say several hundred, each of which is a Minhash. To Minhash a set represented by a column of the characteristic matrix, one needs to pick a permutation of the rows. The Minhash value of any column is the number of the first row, in the permuted order, in which the column has a "1". The probability that the Minhash function for a random permutation of rows produces the same value for two given messages is equal to the Jaccard similarity between those messages. [2]

The expected similarity of two signatures is equal to the Jaccard similarity of the columns. The longer the signatures, the lower the error.

b. Locality Sensitive Hashing

MinHash is an effective technique for reducing the dimensionality of variable-sized alphanumerical data to fixed-length numerical output, with which Jaccard similarity can be probabilistically calculated via random sampling. However, each log hash must still be compared to every other log hash in order to identify relationships, resulting in complexity O(n**2). There are various search methods for identifying nearest neighbors based on space partitioning but they all degrade to linear search for sufficiently high dimensions. Locality Sensitive Hashing is used to reduce the search space by using hash collisions as a proxy for similarity. [3]

c. Clustering using DB Scan

Consider a set of points in some space to be clustered. Let E be a parameter specifying the radius of a neighborhood with respect to a random point in the cluster. For the purpose of DBSCAN clustering, the points are classified as core points, reachable points, and outliers, as follows:

• A point p is a core point if at least a minimum number of points (MinPts) are within distance E of it (including p).

• A point q is directly reachable from p if point q is within distance E from core point p.

- A point “q” is reachable from “p” if there is a path p1, ... ,pn with p1 = p and pn = q, where each pi+1 is directly reachable from pi. Note that this implies that all points on the path must be core points, with the possible exception of q.

- All the points that are not reachable from any other point are outliers or noise points.

Now if p is a core point, then it forms a cluster together with all points (core or non-core) that are reachable from it. Each cluster contains at least one core point; non-core points can be part of a cluster, but they form its "edge" since they cannot be used to reach more points.

Reachability is not a symmetric relation since, by definition, no point may be reachable from a non-core point, regardless of distance (so a non-core point may be reachable, but nothing can be reached from it). Therefore, a further notion of connectedness is needed to formally define the extent of the clusters found by DBSCAN. Two points p and q are density-connected if there is a point o such that both p and q are reachable from o.

5. COMPARATIVE STUDY

Multiple approaches have been taken into account to implement the clustering of the log data and are tested for better performances in terms of accuracy and efficiency. The methods implemented include Minhash-LSH (tested with shingles of variable sizes), and a Graph-based approach in which all the messages are treated as nodes and the weights between the nodes is the Jaccard distance between the messages. K-Means was ruled out from the comparative study because of the constraint with respect to the number of classes that have to be specified in the algorithm. Since the Openstack data set is vast, we do not have the information about the number of classes of the error messages.

a. Shingling

Typically, in the problem of comparison of similar documents, shingle objects of size eight (8-shingles) have been used. Since, we have logs with an average of fifty tokens per log, shingling with a higher number would decrease the precision to which the MinHash-LSH algorithm filters out the similar messages. So to preserve the order of the tokens in the logs, shingles of minimal size 2 have been used.

It has been experimented with shingles of higher-order but the clustering results were not accurate due to the size of the log messages.

b. Non-linearly separable clusters

An algorithm similar to that of DBScan has been used for the clustering of the OpenStack log data. But instead of generating linearly separable clusters of a fixed radius (similar to that of DBScan), we generate non linear clusters as in Figure 7.

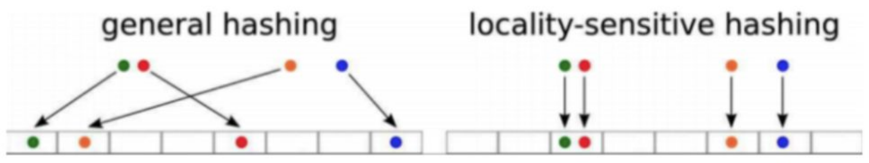


Figure 7: Locality Sensitive Hashing.

The steps used to non-linearly cluster the data points are as follows:

• Take one log in the dataset and using MinHash-LSH to find all the messages that are similar and add them to the bucket.

• For every log in the bucket, repeat step 1 until all messages in the bucket are visited.

• Move to the next non-visited point on the dataset and repeat the previous steps

• By the end we get all the non-linearly separable clusters in the dataset as in the figure

6. Results and future work

The proposed algorithm has been tested on a subsection of data from a single server, which contains approximately 34k log messages, and the achieved the desirable clustering results where all the similar log messages have been clustered into one. Once the different clusters are obtained we use these clusters to extract the principal components from every cluster. A principal component of a cluster is a common feature that uniquely represents the given cluster.

In the case of log message clusters, the principal component for a given cluster is defined as the set of common tokens among all the messages in that cluster. For every new log generated, the task is to classify it into the clusters we obtained from training our models on the data available. For this, the new log is compared with the principal components of the existing clusters and would be classified as a member of the cluster to which it has the closest similarity. The clustering results on the subsection of the data, i.e, 34k log messages, have been pushed to a Github repository[6] and any further changes shall be reflected in the repository itself.

Currently, the idea of extracting the principal component form a cluster is to extract the common tokens from all the messages in a given cluster and these common tokens extracted would be the identifiers of that cluster. This method requires comparing every token of all messages with one another to find out the common tokens that represent the cluster. This would not be a time-efficient solution when the model is trained on large clusters.

Thus, one alternative method that could be proposed is to use Genetic algorithms[7] to generate a regular expression that could represent all the messages in the cluster uniquely. Every new log message would now be compared with the regular expression to see if it fits into the cluster.

A Genetic algorithm could be used to generate the regular expression for the messages that belong to the same cluster briefly as follows:

- Get the distinct tokens from the cluster of messages as genes for the algorithm
- Mutation: Add the Regex special characters[8] to the genes.

- Fitness function: The fitness score is measured by how well the generated regex fits the cluster.
- Crossover: We cross a similar set of messages with one another to produce an offspring that closely represents the pair of messages.
- This offspring is again mutated by checking if the tokens in the generated message could be replaced with the regex special characters.
- This process is repeated until a regular expression is obtained that generalizes all the messages in that cluster.

A prototype[9] of the above-mentioned algorithm is available as a web application for testing purposes for future work.

7. RÉFÉRENCES

- [1] https://en.wikipedia.org/wiki/Finite-state-machine
- [2] Shrivastava2014AsymmetricMH Anshumali Shrivastava and Ping Li, Asymmetric Minwise Hashing ArXiv, 2014, abs/1411.3787
- [3] https://towardsdatascience.com/understanding-locality-sensitive-hashing-49f6d1f6134
- [4] https://en.wikipedia.org/wiki/MinHash
- [5] https://dl.acm.org/citation.cfm?id=997857
- [6] https://github.com/RavicharanN/Rally-Log-data-Analysis/
- [7]https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3
- [8] https://www.regular-expressions.info/characters.html
- [9] http://regex.inginf.units.it/