Mining Telecom System Logs to Facilitate Debugging Tasks

Alf Larsson
PLF System management
Ericsson, Research & Development
Stockholm, Sweden
Alf.Larsson@ericsson.com

Abdelwahab Hamou-Lhadj
SBA Research Lab
ECE, Concordia University
Montreal, Canada
abdelw@ece.concordia.ca

Abstract. **Telecommunication systems are monitored continuously to ensure quality and continuity of service. When an error or an abnormal behaviour occurs, software engineers resort to the analysis of the generated logs for troubleshooting. The problem is that, even for a small system, the log data generated after running the system for a period of time can be considerably large. There is a need to automatically mine important information from this data. There exist studies that aim to do just that, but their focus has been mainly on software applications, paying little attention to network information used by telecom systems. In this paper, we show how data mining techniques, more particularly the ones based on mining frequent itemsets, can be used to extract patterns that characterize the main behaviour of the traced scenarios. We show the effectiveness of our approach through a representative study conducted in an industrial setting.**

Keywords— **System logs, event correlation, troubleshooting of telecom systems, mining algorithms.**

I. INTRODUCTION

Ericsson is one of the largest telecom companies in the world. It has a much diversified product portfolio comprising of various network components. These components work together and are usually distributed in nature. When errors or abnormal behaviours occur, software engineers turn to the analysis of logs, generated by monitoring and tracing the system's activities. Logs, however, tend to be overwhelmingly large, which hinders any viable analysis unless adequate (and practical) tool support is provided [5, 6].

Log analysis is a broad topic and varies in scope depending on the application domain. In this paper, we focus on the problem of extracting meaningful patterns from system logs to help software engineers understand the main behaviour of the traced scenario. The ultimate goal is to facilitate debugging and other maintenance tasks. Consider, for example, the simple scenario of transferring a file over FTP (File Transfer Protocol) between two network sites. The generated log file is bound to noise and network interferences (as it is almost always the case in industrial systems). Knowing which events are most relevant to the file transfer itself is a challenging task. But when performed properly, it can reduce significantly the time and effort it takes to software engineers to understand and troubleshoot the system in case the transfer fails.

At Ericsson, a common approach is to look at the occurrence of events and relate them using timestamp information. Due to noise and interference in the data, this type of analysis has limited ability to uncover correct and complete behaviour. Hence, the process often requires heavy involvement of domain experts.

In this paper, we investigate the use of data mining techniques for identifying and analyzing important events and patterns in large system logs with minimum intervention of the users.

II. APPROACH

Our approach is shown in Figure 1. It encompasses two main phases: Pattern Generation and Validation, and Pattern Matching. The first phase is a learning phase in which we apply data mining techniques to extract behavioural patterns from large logs. The extracted patterns are presented to domain experts for validation. Domain experts can choose to assign a context (a description and any other relevant information) to the valid patterns. The patterns with their description are then saved in a database.

During the pattern matching phase (the second step), we use the pattern database to correlate events in a random set of logs generated from systems in operation using pattern matching techniques. We also support the possibility to correlate these patterns. This is particularly useful if the traced feature involves several scenarios. Software engineers can see how these scenarios are interrelated. These phases are explained in more details in the subsequent sections, preceded with a subsection on log generation.

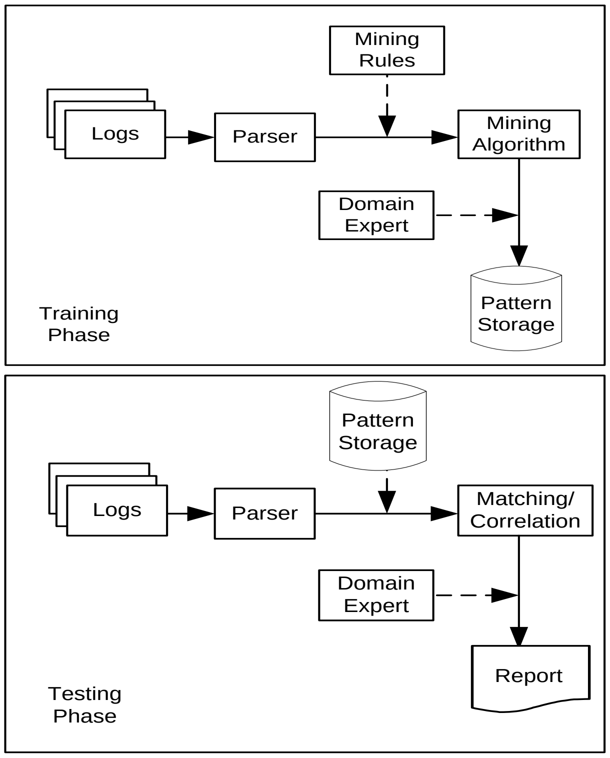


Figure 1.Overview of the approach

A. Collection of System Logs

We generate logs by exercising the system with a usage scenario of interest. Our strategy is to run the scenario several times with different background noise and feed the resulting log files to a data mining algorithm to automatically extract the common sequences of events. Our hypothesis is that the events that are common to the generated log files are the ones that are also the most relevant. This approach is similar to Software Reconnaissance introduced by Wilde et al. in [12] and further improved by many researchers (see [4] for a survey). The authors compared traces generated from routine call traces to identify the components that implement a particular scenario. This contributes to solving a problem known as feature location in code; identifying the most relevant components that implement a specific feature.

The main difference is that we use a data mining algorithm instead of correlating traces using graph theory. Also, to our knowledge, software reconnaissance and other feature location techniques have not been applied to network logs. This type of run-time information differs greatly from traces of control flow. Network logs tend to be more fine-grained and the information cannot be easily mapped to the source code. Many feature locations techniques heavily rely on the source code to find relevant information.

B. *Learning Phase: Pattern Generation*

There exist several data mining algorithms that extract patterns from large data. Examples include Apriori [1], Frequent Pattern tree (FP-tree) [2], FP-Growth [10, 11], etc. The one we chose to use in this paper is MAFIA [3]. MAFIA stands for Mining Maximal Frequent Itemsets Algorithm. We selected MAFIA because of its time efficiency for extracting long patterns compared to its counterparts [3].

MAFIA starts by building a lattice tree that represents the lexicographic ordering of the items in an itemset. We will detail this process in the subsequent paragraphs. The algorithm then applies a depth-first search algorithm with pruning techniques to detect maximal frequent itemsets that have a support greater or equal than a certain threshold. The support of an itemset represents the number of times it appears in the itemset database. To illustrate how MAFIA works, let us revisit the example provided in [3]. In this example, I, the item set, contains four items I = {1, 2, 3, 4}. A database of itemsets, T, is a multiset of subsets of I. The objective of the algorithm is to find the maximal frequent itemsets in T. For example, the result of applying MAFIA to T = {{1234}, {123}, {134}, {234}} is {123} (with minimum support = 2).

The algorithm starts by building a lattice in such a way that the top itemset is the empty set and each lower level k contains all itemsets of length k. The itemsets are ordered according to the lexicographic ordering relationship. The lexicographic subset lattice generated from I is shown in Figure 2. The first level (k = 0) is the empty set. The next level contains itemsets of length 1, ordered in the lexicographic way {1} < {2}, etc. The next level (where the effect of ordering is more noticeable) contains items of length 2. In this level, Itemsets {12}, {13}, {14}, generated from extending {1}, are also ordered in the lexicographic manner, etc.

One simple way to find maximal frequent itemsets is to apply a naïve depth-first algorithm and count the number of occurrences of each itemset. An itemset with a support greater or equal to the minimum support is added to the maximal itemset database (the output of the algorithm), given that a superset has not already been discovered. The problem with a simple depth-first algorithm is that it tends to be unnecessarily slow. This is because it counts the frequency of all itemsets in the tree despite the fact that some subtrees can be quickly filtered out earlier in the process if a different reordering is used. For example, given the subtree rooted at P = {1}, counting the support of {12}, {13}, {14}, P's children, first will reveal that only {12} is frequent (it appears twice in the database). Items 3 and 4 can then be filtered out so no new itemsets need to be counted in the subtree rooted at P. In other words, {123}, {124}, and {234} do not need to be counted which will save time. Based on this, the MAFIA authors [3] present various pruning and reordering algorithms to increase the performance of the algorithm by reducing the search space. An implementation of MAFIA is made publicly available by the authors on http://himalaya-tools.sourceforge.net/Mafia/.

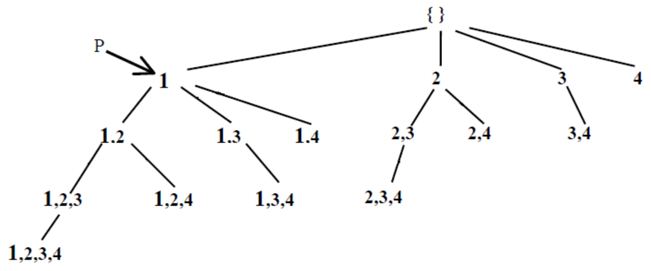


Figure 2. Lexicographic ordering lattice (from [3])

To apply MAFIA to log events, we introduce the following definitions. We call an instance of a given scenario a '*window*'. We distinguish each instance with a unique window identifier (window_id). We save the generated logs in a database table. Each column consists of a specific attribute of a log event. We assign a unique integer transaction id to each distinct attribute. The idea is to use the ids for lexicographic ordering. While assigning transaction ids, window limit is not taken into consideration. Even if a log entry is found across multiple windows, it is assigned the same transaction id. Figure 3 shows an example of two scenarios, represented as windows, with their events (depicted using A, B, C…). The event attributes are assigned transactions ids. In this example, I = {1, 2, 3, 4, 5, 6, 7} and T = {{121234}, {1251267}}. The result of the algorithm (with minimum support = 2), is Itemset {12} which corresponds to the pattern AB.

Window ID	Window 1	Window 2
Sample logs for each window	A B A B C D	A B E A B D F G
Transaction ID	1 2 1 2 3 4	1 2 5 1 2 6 7

Figure 3. Example of scenarios represented as windows

We want to note that, in order to obtain a pattern representing a specific behaviour, we do not consider all attributes of a log event during the mining process. For example, a typical log event generated from file transfer operation will involve the timestamp, protocol, sender's IP, receiver's IP, and other information. Considering the timestamp will end up eliminating the very possibility of obtaining a pattern by making each event unique because timestamp varies from one event to another. The decision on which attributes to select is left to the user. It is always recommended to use attributes that represent a generalized behaviour. The more attributes we use, the more restrictive the pattern detection approach is.

C. *Pattern Validation and Context Assignment*

Once we extract the patterns, we present them to domain experts at Ericsson for validation. This is usually done in a semi-automatic way using a tool we have developed for this research (see the case study for snapshots). A typical task of the domain expert is to go through the pattern, assess its quality, and remove unnecessary data if need be. There might be situations where the domain expert considers the quality of the pattern to be poor (e.g., it lacks key events). In this case, he or she can request either to re-run the pattern mining process by adjusting its input (adding other attributes) or run the scenario again with additional background noise to clearly distinguish the behavioural pattern events from other events.

Once the domain expert deems the pattern to be valid, he or she assigns a context, which is a high-level description comprised of the pattern name, the context in which the pattern appeared (e.g., the network topology, the type of communication used, the communication protocol, etc.). The pattern is then saved in the pattern library.

D. Testing Phase: Pattern Matching

We have developed a simple pattern matching algorithm to identify the event patterns in logs generated from a system in operation. The algorithm simply matches the log events to the patterns in the database. Our matching algorithm operates as follows: Given a sequence of events s1 and a pattern p1 (in the pattern database), s1 and p1 are considered similar if all events in p1 appear in s1. In other words, it is enough for s1 to contain the events that appear in a pattern to be considered as a candidate pattern. An alternative solution will be to consider exact matching but this would turn out to be too restrictive because of noise in the data. Future studies should focus on measuring similarity based on a certain threshold.

E. Testing Phase: Pattern Correlation

We define pattern correlation as the process of identifying a relation between the extracted patterns in a given scenario or a set of scenarios. This task is important for debugging and performance analysis since it can help software engineers identify what is happening in the system when multiple operations occur (e.g., sending an FTP file while at the same time using HTTP).

Patterns are correlated using two methods: attribute-based correlation and time-based correlation. In the attribute-based correlation method, the event attributes are used to find relation among patterns. For example, if a user wants to know which events happened in two patterns on a particular IP address, an attribute-based filtering mechanism can be employed to identify those events. The resulting output will contain only those logs from both patterns which belong to the selected IP. The time-based correlation (the second method) allows users to see which patterns appear within a particular timeframe.

III. Evaluation

A. Target System

We chose CPP (Connectivity Packet Platform) as the target system, which is a proprietary carrier-class technology developed by Ericsson [9]. It has been positioned for access and transport products in mobile and fixed networks. Typical applications on current versions of CPP include third-generation nodes RBSs (Radio Base Stations), RNCs (Radio Network Controllers), media gateways, and packet-data service nodes/home agents. CPP was first developed for ATM (Asynchronous Transfer Mode) and TDM (Time Division Multiplexing) transport.

A CPP node contains two parts, an application part and a platform part. The application part handles the software and application-specific hardware. The platform part handles common functions such as internal communication, supervision, synchronization, and processor structure.

B. Usage Scenarios and Log Generation

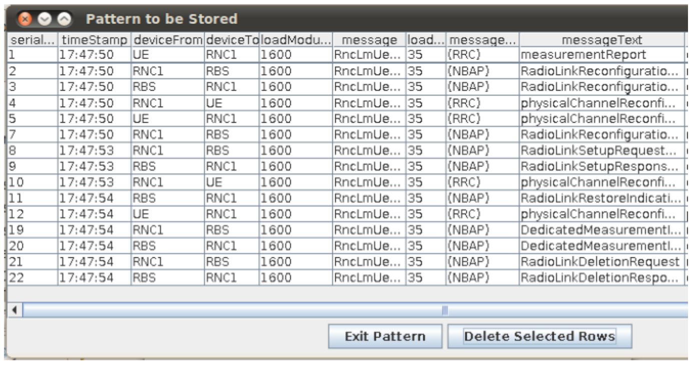
We experimented with various scenarios. Due to the proprietary nature of the system, we choose, in this paper, to present two scenarios: inter-frequency handover (IFHO) and the setup of the radio access bearer (RAB). The results are representative of our findings. Each of these scenarios was performed across various RBSs, where each RBS was working on a set of cells available giving a wide variety of logs.

The log generation tool we used in this paper is the Trace and Error (T & E) package, which is a built-in capability in CPP, used often by software engineers to integrate, verify, and troubleshoot CPP applications [9]. T&E supports two functionalities: the tracing functionality and the error handling functionality. The tracing functionality helps the system and functional behaviors to be traced and reported at software development. The error functionality helps to log fault conditions. The T & E log shows a history of recorded trace and error events on the system.

C. Learning Phase: Pattern Generation and Validation

Scenario 1: IFHO

To generate pattern for IFHO, we run the scenario several times with different background noise. We generated a log file for each run. The size of log files varies from 3 to 7 GB. An IFHO event has many attributes including the event ID, DeviceFrom, DeviceTo, LoadModule, Message, MessageType, MessageText, and Parameters. We fed the log files to the pattern generation component of our approach. We selected the attributes "MessageType" and "MessageText" as the main attributes for the pattern generation process.



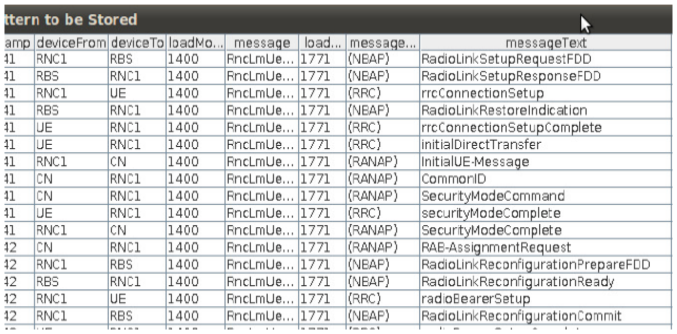
serial...	timeStamp	deviceFrom	deviceTo	loadModu...	message	load...	message...	messageText
1	17:47:50	UE	RNC1	1600	RncLmUe...	35	(RRC)	measurementReport
2	17:47:50	RNC1	RBS	1600	RncLmUe...	35	(NBAP)	RadioLinkReconfiguratio...
3	17:47:50	RBS	RNC1	1600	RncLmUe...	35	(NBAP)	RadioLinkReconfiguratio...
4	17:47:50	RNC1	UE	1600	RncLmUe...	35	(RRC)	physicalChannelReconfi...
5	17:47:50	UE	RNC1	1600	RncLmUe...	35	(RRC)	physicalChannelReconfi...
7	17:47:50	RNC1	RBS	1600	RncLmUe...	35	(NBAP)	RadioLinkReconfiguratio...
8	17:47:53	RNC1	RBS	1600	RncLmUe...	35	(NBAP)	RadioLinkSetupRequest...
9	17:47:53	RBS	RNC1	1600	RncLmUe...	35	(NBAP)	RadioLinkSetupRespons...
10	17:47:53	RNC1	UE	1600	RncLmUe...	35	(RRC)	physicalChannelReconfi...
11	17:47:54	RBS	RNC1	1600	RncLmUe...	35	(NBAP)	RadioLinkRestoreIndicati...
12	17:47:54	UE	RNC1	1600	RncLmUe...	35	(RRC)	physicalChannelReconfi...
19	17:47:54	RNC1	RBS	1600	RncLmUe...	35	(NBAP)	DedicatedMeasurementI...
20	17:47:54	RBS	RNC1	1600	RncLmUe...	35	(NBAP)	DedicatedMeasurementI...
21	17:47:54	RNC1	RBS	1600	RncLmUe...	35	(NBAP)	RadioLinkDeletionRequest
22	17:47:54	RBS	RNC1	1600	RncLmUe...	35	(NBAP)	RadioLinkDeletionRespo...

Figure 4. The IFHO extracted pattern

A domain expert at Ericsson analyzed the resulting pattern and removed some events including repeated signals and heartbeat type messages. These events are considered as noise and can occur at any instant. It took around one hour for the domain expert to clean up the automatically extracted pattern. We believe that this step could be automated (at least at a certain extent) in the future by studying what constitute noise in such systems and build a predefined list of events that can be removed before applying the mining algorithm. We do not expect, however, to completely discard the domain expert from the process. In fact, we believe that domain expert feedback is very useful during the whole process. The final pattern for IFHO consists of 15 events as shown in Figure 4 (note that we do not show some of the event attributes to save space). The pattern is then saved in the pattern database under the name IFHO pattern.

Scenario 2: Radio Access Bearer (RAB) Setup

We followed the same process as for the previous scenario. We run RAB several times with various background noises. The size of the log files varies from 4 GB to 7 GB. The pattern mining algorithm generated a pattern. We gave this pattern to a domain expert who (as before) removed additional data (considered as noise). The resulting pattern contains 31 events and it is partially shown in Figure 5. The pattern is then saved under the contextual name: RAB set-up.



ttern to be Stored

amp	deviceFrom	deviceTo	loadMo...	message	load...	message...	messageText
41	RNC1	RBS	1400	RncLmUe...	1771	(NBAP)	RadioLinkSetupRequestFDD
41	RBS	RNC1	1400	RncLmUe...	1771	(NBAP)	RadioLinkSetupResponseFDD
41	RNC1	UE	1400	RncLmUe...	1771	(RRC)	rrcConnectionSetup
41	RBS	RNC1	1400	RncLmUe...	1771	(NBAP)	RadioLinkRestoreIndication
41	UE	RNC1	1400	RncLmUe...	1771	(RRC)	rrcConnectionSetupComplete
41	UE	RNC1	1400	RncLmUe...	1771	(RRC)	initialDirectTransfer
41	RNC1	CN	1400	RncLmUe...	1771	(RANAP)	InitialUE-Message
41	CN	RNC1	1400	RncLmUe...	1771	(RANAP)	CommonID
41	CN	RNC1	1400	RncLmUe...	1771	(RANAP)	SecurityModeCommand
41	UE	RNC1	1400	RncLmUe...	1771	(RRC)	securityModeComplete
41	RNC1	CN	1400	RncLmUe...	1771	(RANAP)	SecurityModeComplete
42	CN	RNC1	1400	RncLmUe...	1771	(RANAP)	RAB-AssignmentRequest
42	RNC1	RBS	1400	RncLmUe...	1771	(NBAP)	RadioLinkReconfigurationPrepareFDD
42	RBS	RNC1	1400	RncLmUe...	1771	(NBAP)	RadioLinkReconfigurationReady
42	RNC1	UE	1400	RncLmUe...	1771	(RRC)	radioBearerSetup
42	RNC1	RBS	1400	RncLmUe...	1771	(NBAP)	RadioLinkReconfigurationCommit

Figure 5. The RAB set-up pattern

D. *Testing Phase: Pattern Identification and Correlation*

Once we identified the patterns, we used them to find patterns in the system during operation. For this purpose, we started by correlating patterns based on their attributes. To do that, we needed a log file which had a combination of scenarios. We chose a scenario that combines a set of different telecommunication sub-scenarios including IFHO, soft handover, softer handover, RAB set-up, etc. We run the scenario as it would be in real world (i.e. with background noise). The generated log file was fed to the pattern matching and correlation component of the framework. We were able to automatically identify the IFHP and RAB patterns using the pattern database built during the learning process.

Once we had the pattern highlighted, we were able to use attribute and time based correlation techniques to gain insight into what is happening in the scenario. For example, we were able to identify the most frequent destination site for IFHO messages. We conducted similar experiments using time correlation by identifying the patterns that occur within a specific timeframe. We needed for this task to do some preprocessing steps to align the time generated from parallel systems. The correlation, in this case, showed all complete patterns obtained for IFHO and RAB between these time intervals. This was helpful for following the flow of messages exchanged between different network sites.

We have shown the results to Ericsson software engineers working on troubleshooting tasks. The feedback we received shows that the approach holds real promise in simplifying the analysis of telecommunication logs, and reducing the time and effort spent on understanding their content.

IV. Conclusions

In this paper, we demonstrated the potential of using data mining techniques, more particularly the MAFIA approach, to extract useful information from telecom logs. The objective is to help software engineers analyze these logs more efficiently and precisely. Based on the feedback we received from software engineers at Ericsson, the approach is helpful and promising. In particular, they report that this technique (1) reduces the manual effort put into indentifying relevant events required for debugging, and (2) increases the precision of relevant event identification. As future work, we intend to continue exploring the application of data mining approaches to alternative types of log analysis. We will also investigate what constitute noise in this type of data to further reduce the time spent by domain experts to identify patterns. Some techniques that can be useful to explore in this context are the ones presented in [7], in which the authors discuss the impact of utilities (noise) on the size of traces.

Acknowledgment: Thank you to software engineers at Ericsson, Stockholm, for active participation and providing continuous feedback.

V. References

[1]. R. C. Agrawal, T. Imielinski, and R. Srikant, “Mining association rules between sets of items in large databases,” *In Proc. of the ACM International conference on Management of data*, pp. 207-216, 1993.

[2]. R. C. Agarwal, C. Aggarwal, and V.V.V. Prasad, “A tree projection algorithm for generation of frequent itemsets,” *J. of Parallel and Distributed Computing*, pp. 350–371, 2001.

[3]. D. Burdick, M. Calimlim, and J. Gehrke, “MAFIA: A maximal frequent itemset algorithm for transactional databases,” *In Proc. of the 17th International Conference on Data Engineering,* pp. 443 - 452, 2001.

[4]. B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature Location in Source Code: A Taxonomy and Survey," *Journal of Software: Evolution and Process, 25(1),* pp. 53–95, 2013.

[5]. A. Hamou-Lhadj, "Techniques to Simplify the Analysis of Execution Traces for Program Comprehension," *Ph.D. Dissertation, School of Information Technology and Engineering (SITE),* University of Ottawa.

[6]. A. Hamou-Lhadj, and T. Lethbridge, "Measuring Various Properties of Execution Traces to Help Build Better Trace Analysis Tools," *In Proc. of the 10th International Conference on Engineering of Complex Computer Systems*, pp. 559-568, 2005.

[7]. A. Hamou-Lhadj, and T. Lethbridge, "Reasoning About the Concept of Utilities," *ECOOP International Workshop on Practical Problems of Programming in the Large, LNCS, Vol 3344, Springer-Verlag,* pp. 10-22, 2004.

[8]. G. Jakobson, L. Lewis, and J. Buford, “An Approach to Integrated Cognitive Fusion,” *In Proc. of 7th International Conference on Information Fusion,* 2004.

[9]. L-Ö. Kling, Å. Lindholm, L. Marklund and G. B. Nilsso, “CPP. Ericsson Review No. 2”, 2002. Available online:http://www.ericsson.com/ericsson/corpinfo/publications/review/2002_02/files/2002023.pdf

[10]. B. S. Kumar and K.V.Rukmani, “Implementation of Web Usage Mining Using APRIORI and FP Growth Algorithms,” *Journal of Advanced Net and App, 1(6),* pp. 400-404, 2010.

[11]. Y-C. Li, C-C. Chang, “A New FP-Tree Algorithm for Mining Frequent Itemsets,” *Springer Lecture Notes in Computer Science Volume 3309,* pp 266-277, 2004.

[12]. N. Wilde and M. Scully, "Software Reconnaissance: Mapping Program Features to Code," *Journal of Software Maintenance: Research and Practice, 7(1),* pp.49-62, 1995.