International Journal of Computer Applications (0975 - 8887)
Volume 124 - No.5, August 2015

RecB: Set Theory based Technique for Large Scale Pattern Mining in Web Logs

Tanya Steen, Ray Lindsay
Enterprise Analytics
Australian Taxation Office

ABSTRACT

Web Analytics is a way of turning raw data into actionable information. Large organisations own web based applications and connect to external databases which generate very large web logfiles. It then becomes crucial to estimate how information systems are accessed by staff, what their search preferences are, what documents are of greater demand. One challenge in obtaining this knowledge is that logfiles contain unstructured information where authentic search requests are not discriminated from crawler hits. Another challenge is that many proposed pattern mining techniques are usually tested on small benchmark datasets, so their performance on a large scale is hard to predict. This paper stresses the importance of data preprocessing and introduces an efficient method for mining patterns in large sized collections of web logs (of all types) based on classic set theory properties.

General Terms

Algorithms, Big Data

Keywords

set theory, pattern mining, web mining, computational complexity, complexity reduction, big data analytics

1. INTRODUCTION

Set theory has often been regarded as a universal theory behind mathematics, whose role was to generalise our reasoning. Set theory is integrated into many branches of mathematics and computer science, thus discovering new set-theoretic properties has often been inspired by a question from another field, helping to find new applications of that theory to other areas. This paper describes some properties of set theory applied to web usage mining and offers a method for pattern identification in weblog data. As such, the problem of mining web logs was attended to years ago and a number of techniques was introduced. In general, there are three major knowledge discovery domains that pertain to web mining: content mining, user path tracking and web usage. While in the first two areas there has been significant progress, in the area of estimating web usage some obstacles remain. One major obstacle in that area is the difficulty in discriminating between authentic user requests and automated search engine responses all generated at the same time and written in web logfiles in the same manner.

Traditional user activity analysis is often centred around traffic traces collected at user level at aggregation points inside the network. This is a very straight forward hardware related methodology to capture user activity. One metric for usage comparison was proposed in [1] and is the proportion of connections corresponding to a particular port, that is the ratio of connections on one port to the total number of connections. However, if an organization is subscribed to commercial (external) databases, it usually uses the shared IP address for the whole office, which makes it impossible to trace individual user traffic.

Web servers register and collect records about interactions between servers and users every time a search request is initiated. Web access logs can help to understand the user search preferences and the web structure, thereby improving the design of such a complex system. To do this, web usage mining is performed in the two directions: general pattern mining and pattern mining at user level. The general pattern mining analyses the web log data to understand access patterns and trends. Such analyses can shed light on better structure of web resources. Patterns at user level help to understand and measure one's performance and evaluate access to information. However, quite often the availability of software for this purpose has serious limitations and is therefore unsatisfactory, according to [2].

Web log records generally contain information about users and their activities thus the study of such data can help to profile clients, their search preferences and estimate volumes of information they retrieve. Log records are essential to understand the activities of complex systems, particularly in the case of applications with little user interaction such as server applications. In most cases, log files are too verbose and hard to understand; they need to be subjected to log analysis in order to make sense of them. This also requires sophisticated log analysis software, usually tailored to specific organisational needs.

The popular free downloadable open source Web analytics tools are listed in the relevant wikipedia page under the title *List of web analytics software*. Tools such as these typically analyse web server log files, extracting items such as visitors' IP addresses, URL paths, access times, user agents, referrers, etc. and grouping them in order to produce HTML reports.

Despite many obvious advantages of using free Web analytics tools, there are some drawbacks. The most common criticisms of such tools include:

—Generated statistics do not discriminate between humans and robots (a.k.a. crawlers). Web crawlers are mainly used to create a copy of the visited pages for later indexing to speed up access. Crawlers can also be used for automating maintenance tasks like checking links or validating HTML codes. Crawler hits are normally counted as visitor hists. As a result all reported metrics are higher than those initiated by humans only. Some tools produce unrealistic figures of visits, which may be many times higher than the data produced by JavaScript based web statistics.

—Query string analysis is not performed. Consequently, dynamic generated websites can not be listed separately (e.g., PHP pages with arguments)

JavaScript based programs have their limitations, too. The potential impact on data accuracy comes from users deleting or blocking cookies. Without cookies being set, the JavaScript code will not be able to collect data.

Another limitation of the majority of web analytics tools is the use of sampling in statistical computation. On the one hand, it reduces the load on the servers, but on the other hand, the generated reports are limited by the size of a sample. If hits (not sessions) are sampled, it is difficult to estimate the margins for various types of errors. The above listed and other limitations in using popular web analytics tools led to the development of new approaches to web usage mining.

Many of the developed web log mining techniques work around growing search or prefix trees. While trees generally serve as compact structures to effectively store and access data, it may be costly to actually build these structures in large datasets. Most of the publications on the subject demonstrate experimental results obtained on small datasets, so the full potential of the proposed algorithms could not be seen. That uncertainty could be cleared by analysis of the algorithms' worst-case computational complexity as this is the only way to guarantee an upper bound on the computational time.

Complexity analysis has demonstrated its practical value by giving us a better understanding of the algorithmic difficulties of a problem, which affect how much of the computational power is required to mitigate the performance time. Theoretical estimates of computational complexity could be a good practical guide to algorithm's structure and efficient computational methodologies.

The next sections will address the challenges associated with diversity of web logs and their filtering, describe the most notable developments in the area of web log mining to better understand the purpose of the technique *RecB*, which is introduced later on. The computational complexity analysis of *RecB* and popular tree based methods will then further strengthen our argument about what type of methodology can be suitable for mining large collections of log records.

2. LOG RECORDS

The log files often provide mixed data on using internal and external databases, and one challenge in actually processing the data is that a single search request triggers automated requests by a *database engine* (a document retrieval system) generating up to hundreds of logs. In order to obtain accurate information about user activity one has to find ways of discriminating between initial requests and automated responses. Figure 1 illustrates the interaction (requests and responses) within a typical information system and generation of log files.

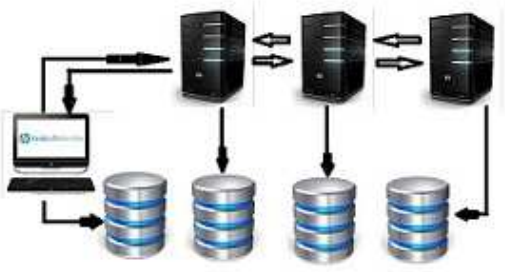

Fig. 1 Typical information system with a desktop, web server, proxy servers and web logs

The log files suitable for Web usage mining usually come from three different locations: Web server, Web proxy server and client browser. Web server logs provide complete data on usage but do not keep record about cached pages. Web proxy server processes HTTP requests, sends them to Web server, and only then the result is returned to user. Proxy servers provide an intermediate level of caching. Multiple users could view the cache generated by a single search request. Besides, all requests from a proxy server share the same ID, which makes it difficult to identify individual users. These and other problems have been addressed in [3][4].

Storing log records in multiple places makes data mining for navigational patterns quite challenging. For example, server logs do not contain Web access data, which is cached by proxy servers or on the client side. Log records from all three sources complement each other, thus must be joined together to maintain accuracy. This can be hard to accomplish, which explains why many web log mining algorithms are designed to work only with one type of log records (usually server side) and not with all types.

Web log files vary in size depending on the period of time they cover and may reach hundreds of Gigabytes. A significant proportion of web logs is usually redundant thus the initial step is to filter the redundant logs out, which is a big problem in itself.

Table 1 illustrates the diversity in log file formats. Note, the names, links and numbers in this table are artificial.

2.1 Query Analysis

For query analysis, the scripting language *Python* was used because of its string processing capability. *Python*'s built-in string classes support the sequence type methods and also the string-specific methods. There are seven sequence types: *strings, Unicode strings, lists, tuples, bytearrays, buffers* and *xrange* objects. All sequence types support comparisons. In *Python*, strings are *immutable sequences* (i.e., objects whose value is unchangeable once they are created). *Python* code treats elements or substrings just like it treats a sequence. *Python* refers to substrings with the flexible *slice* operation.

Type of server	Log Sample
elib	20110224150925+1100(1298560165) Get /datastora gex/view.htm?docid=AUS/AUS2003377/D00001 67.999.112.88 Mozilla%2F5.0%20 (compatible%3B%20Yahoo!%20Slurp%3B%20 http%3A%2F%2Fhelp.yahoo.com%2Fhelp%2Fus%2Fysearch%2Fslurp) %0.227 3 84.501 0.545
idatalib	07/22/11 14:05:40 10.99.99.66 user7856 Allowed none 23 21.9KB 43.5KB www.databasez.com.cn/search/default.aspx?st=Shanghai+Textile+Fabric +and+Cotton+Co+Ltd&srtid=0
wiki	user023 1/06/2010 9:47 ewiki_LIB_acronyms_and_abbreviations_overview
wiki	user023 1/06/2010 9:47 ewiki_LIB_acronyms_and_abbreviations_working_taxonomy
External DB	”www.datareference.com”, ”Reference”, ”75046”, ”12757”, ”530225529”
External DB	”www.datareference.com.ezproxy1.acu.edu.au”, ”Education”, ”99”, ”6”, ”404958”
Internal DB	”user66”, ”QLD/0053146971/D00008”, ”20120416191838”

Table 1. Diversity in log records.

There are several ways of composing string literals in *Python*. One can use single or double quotes, and there are other variations on quoting that are useful. The *Python* module *string* offers functions that transform strings in common ways and can be combined to perform uncommon transformations. For example, the functions .split() and .join() provide a quick way to convert between strings and tuples, which is useful when working with diverse types of log records. The module re helps to identify regular expressions in textual parts of log data. Simple regular expressions can be concatenated to form more complex regular expressions. Additional information on these and other string methods may be found in the *Python Reference Manual*.

While the log files generated by information systems are usually of structured design, the textual information within a record is often unstructured. For example, refer to the first two cases in Table 1. The search activities via Yahoo! Help and databasez.com are represented in the log file differently. In the first record, the useful for analysis part of the string is represented by the document path "AUS/AUS2003377/D00001", in the second - by the key words used for search (*”Shanghai”, ”Textile”, ”Cotton”, ”and”, ”Wool”*).

Clearly, differentiated approaches need to be applied to processing log records with unstructured textual information. For instance, in the first record, the pattern "docid" is present, which always prefaces the path to a specific document. In the second record, there is the term "search", which indicates the presence of key words separated (in this case) by the symbol ”+”. These and other identification criteria can be implemented with *Python* string processing functions in order to define the category for an information request, its weight and origin (i.e., human or non-human). Different sections of the document path separated by slash offer valuable information about the search domain and subdomains, which also contributes to the quantitative part of analysis.

The next pair of records in Table 1 is the initial request from the *user023* for the *Acronyms and Abbreviations Overview* and an unspecified request from an unspecified client. In the log file, the first case is represented by the original query and series of very similar records with the same timestamp indicating that several webpages have been retrieved. In the second case, no any information is available about who initiated the search and what exactly it was about, although several responses from datareference.com had been triggered. Table 1 stores just one such a record - the one containing the pattern "proxy", for demonstration purposes, to show that in this case it would be relatively easy to discriminate the authentic user request from automated responses generated by the database engine.

Often enough, log records do not contain identical timestamps or system specific terms, thus other (domain specific) criteria are required to filter the crawler hits out with full certainty. Once found, such a criteria form the framework for an automated log file processing. With this scenario, the computational part of the framework will consist merely of splitting and pattern matching operations on strings and lists, for which *Python* offers a powerful capability.

2.1.1 Example. The following example demonstrates how *Python* function .findall() represents a line of text with punctuation as a set of terms in just one go:

```
>>> import re
>>> phrase = "Hello, world!"
>>> print re.findall(r'\w+', phrase)
['Hello', 'world']
```

2.2 Data Cleaning

The important aspect of web usage mining is data cleaning and preprocessing. The approach proposed in this paper (the algorithm *RecB*) is designed to work with authentic search requests. Some techniques do not make that distinction. For example, the WEKA webmining tool considers all log records and, as a result, does not accurately aggregate relevant data at user level, as outlined in [5]. Amongst the popular publications on web mining techniques and applications only few, namely [3][6-8], touch on the subject of 'data cleaning' by identifying a 'single user session'. The above authors view an authentic user session as an isolated in time set of logs.

While this approach may work well on small web resources, in large organisations it can be rather challenging. Users or work

areas may share the same IP address, and information systems can be too complex and generate too many logs to fit into the predefined timeframe. For this reason, it would be sensible enough to not rely on timestamps but rather focus on mining logdata in order to identify patterns that help to single out unique user requests.

The problem of identifying user sessions has been discussed and some methods proposed in [3][9][10]. As mentioned earlier, logfiles come in various types and formats and from various locations. For instance, encrypted transactions or interactions do not get combined with related logs on the same resource, thus do not allow to form the logical whole associated with a user. One heuristic for user identification is to use the server access log in conjunction with the referrer log. Another heuristic is to use path completion methods, where the site topology is also used to check the number of links to the requested page. Timeouts could also be used to brake series of records into separate sessions, if logs span long periods of time.

In the data cleaning step, a log set is examined and items such as media files, executable *cgi* files and alike are removed. Log entries of the HTML file requests could be relevant thus should be kept for a more detailed analysis. Items associated with HTTP errors, status codes such as *400* or *500*, crawler signatures are removed. Records with extensions such as *gif, jpeg, jpg, mpg* are removed selectively. Why some graphic files could be retained is explained in detail in [3].

The remaining logs should contain only fields suitable for pattern mining. These usually are the user ID, timestamp (optional), current webpage visited or current document accessed, referrer, etc. Then the data is converted into a format required by the mining algorithms. As suggested in [4], for one's convenience, it may be worthwhile to store the logs as two sets — transactional set and sequential set. The transactional set is defined by the two fields — record ID and the set of documents (pages) accessed. A sequential set contains lists of documents accessed over period of time and the server data.

3. WEB USAGE ANALYSIS

Once the redundant log content has been filtered out, the remaining search requests have to be aggregated at user or group level, interpreted and categorised into groups or topics of interest according to some categorisation principle. In practice, it translates into processing large amounts of records containing combinations of items in order to establish what category they belong to and if the topic in question is popular. To do this, all records should be visited and the combinations of relevant items explored and counted, which often implies exponentiality in computational complexity.

Many efforts have been made towards developing mining techniques for web log data. They are usually based on the apriori paradigm or tree building methodology, such as Web Access Pattern (WAP) tree, initially introduced in [11]. A new data structure (WAP-tree) is built to compactly store sequences of items and corresponding counts. The tree also maintains linkages between prefixes and suffixes. The actual mining takes place after such a tree has been devised, to boost scalability. The total number of scans required to grow a WAP-tree is usually two. The recursive procedure is then applied to enumerate access patterns from the tree performing conditional search. The algorithm counts frequent events in the set of prefixes with respect to condition as suffix, to find *all* Web access patterns.

The Combined Frequent Pattern Mining (CFPM) algorithm proposed in [12] also grows a tree similar to FP-tree or WAP-tree. The main difference is that the improved tree has no header table as such but instead has indices for tree nodes at same level. The improved tree is frequency-ascending and contains more single paths than the frequency-descending original. It also has more branches at its higher levels than the original. A branch is removed after it has been processed and, as a result, the total number of nodes increases at a slower rate which helps to optimise the overall performance.

The State Machine (SM) tree and the Pattern Discovery (PD) tree techniques introduced in [7] also work around tree like structures in order to optimise pattern mining process. The central idea of the *SM-tree* and *PF-tree* algorithms is to test the subsequence inclusion to ensure that the input data sequences are visited just once. The theoretical basis for this approach is a deterministic finite automation (DFA), or deterministic finite state machines. Finite automata constitute a perfect illustration of basic concepts in set theory. SM-trees or PD-trees are built via joined automations which scales the computation of candidate itemsets. The algorithms do not require more memory in large datasets, but their performance does depend on the number of variables, like in the WAP-tree based techniques.

The Breadth-First linked WAP-tree (BFWAP-tree) algorithm was introduced in [13]. The algorithm is designed to mine frequent itemsets that contain information on parent-child relationship of nodes. *BFWAP* grows the frequent header node links of the original WAP-tree in a Breadth-First fashion and then examines each node in order to check the parent-child relationships between nodes. After that it computes frequent sequential patterns through progressive Breadth-First sequence search. The proposed algorithm does not re-construct the WAP-tree, which helps to speed up its performance.

The *mWAP* algorithm proposed in [14] modifies a WAP-tree structure, so that reconstructions of the intermediate WAP-trees are not required. A binary code is used to mark each node in the modified tree structure. These binary codes help to identify the position of the nodes in the tree during mining. The nodes are then linked to keep track of those with the same label of prefix sequences. The algorithm scans the database two times. It also builds a prefix tree data structure by inserting the frequent sequence of each record in the tree just like the WAP-tree algorithm. After that, the tree is visited again to build links between the frequent header nodes.

The Sequence Tree (ST) algorithm introduced in [4] is designed to identify the frequent sequences of items in web logs. The *ST* reads the input data and creates a map with key-value pairs. Key refers to the unique web page that was visited and sequence value refers to the number of visits. Like other tree building techniques, the *ST* consists of the two stages - building a sequence tree structure and then mining that tree in order to find patterns of interest.

A novel algorithm called *Bidirectional Growth based mining Cyclic behavior Analysis of web sequential Patterns* (BGCAP) has recently been introduced in [15]. The BGCAP combines Web

prefetching with properties of directed acyclic graphs in order to reduce the levels of recursion during mining stage. The authors also provide an extensive overview of pattern mining algorithms for web logs.
According to [7][8], in the web usage mining process, the data mining techniques are applied in order to identify the trends and the patterns in the ways users browse websites. Figure 2 illustrates the major stages of that framework.
The navigation patterns extracted from log records can help in optimising the structure of the website. Thus the stages of pattern discovery and pattern analysis are absolutely essential in web usage mining. The main approaches to pattern discovery in web log data suggested in [7][8] are:

(1) Association Rules (to predict the correlation of items)
(2) Sequential Patterns (to discover users navigation behaviour)
(3) Cluster Analysis (to profile similar items.)

The framework introduced in the next section can power all three approaches because it helps to efficiently identify and classify patterns of interest by eliminating redundant itemsets from data.

4. THE E-RECORD BREAKING ALGORITHM RECB

The record breaking algorithm *RecB* is the technique designed to rearrange and compress electronic records down to subrecords in order to reduce search space for further categorisation. First, it finds the frequent subrecords (or itemsets) and substitutes duplicates with a single set of items. The modified structure then contributes to Content Analysis performed to estimate a user activity, user preferences and information systems usage overall. With the *RecB* this is done more efficiently because the algorithm focuses on elimination of records, or their sub-records, and does not grow new data structures like the WAP-tree based methods.

4.1 Substitution of eRecords by Unions of Subrecords

The central idea of this space reduction method is that a record can be substituted by its subrecords according to the following principle:
If a set of items of cardinality M contains the infrequent 2-itemset AB, it can be substituted by 2 subsets of size M-1, such that one subset will not contain the item A and the other will not contain the item B.
Definition 1. *If an object* $o = \{i_1,...,i_m\}$ *contains one infrequent 2-itemset* $\{i_p, i_q\}$, *then o can be substituted by its two subsets*
$s_1 = \{i_1,...,i_{p-1},i_{p+1},...,i_m\} = o \setminus \{i_p\}$ *and*
$s_2 = \{i_1,...,i_{q-1},i_{q+1},...,i_m\} = o \setminus \{i_q\}$ *such that* $s_1 \cup s_2 = o$ *and* $o \longmapsto \{s_1, s_2\}$. *We will call the subsets* s_1 *and* s_2 *projections of the object o.*

Now o is substituted by the union of its subsets s_1 and s_2. If a record contains one infrequent *2*-itemset, it is substituted by its two sub-records. If it contains one infrequent *3*-itemset, it can be substituted by its *3* sub-records, and so on. The number of projections resulted from one infrequent k-itemset is, therefore, defined by k and is bounded by $\binom{m}{2}$. Thus for the present let it suffice to consider the infrequent *2*-itemsets in order to better understand a computational part of the method.
An infrequent itemset of size k effectively divides an object into a set of smaller subsets. The number of projections resulted from division with one infrequent k-itemset is k. The number of projections resulted from division with the second, third, etc, infrequent k-itemsets can vary. It depends on the k-itemsets, particularly, if they contain the items already used in the previous divisions. If the k-itemset picked for breaking does contain at least one such an item, the number of projections after division will be smaller. The number of projections also depends on the difference between m and k. The greater this difference is, the more *(k+1)*-itemsets can be identified in a record, therefore, the more divisions will be performed. Finally, the number of projections depends on the number of infrequent itemsets the record contains.
It is important to note that projections should not be of length equal or less than the length of the infrequent itemset by which they are divided. It is also important to note that the support of an itemset contained in two or more projections of the same object is the same as that of the record.

As a part of pre-processing, the dataset is scanned and the infrequent *1*-itemsets are identified and deleted. The items are lexicographically ordered. The data records are placed into a hash table, so the computations are performed on the unique records only. Once the dataset is ready, the *RecB* algorithm makes k paths through the data. The value of k is incremented by *1* from *2* to, say, P, which is usually not large. The algorithm computes only the itemsets of length k at a time. This is done because the number of k-itemsets in a record of length m can not be greater than $\binom{m}{k}$. To do this is more effective than to generate all subsets of an m-itemset, excluding the empty set and the m-itemset.
In each step, *RecB* computes all k-itemsets and stores them in a hash table. It then tests each k-itemset against the predefined threshold. After this procedure, the frequent and infrequent k-itemsets are stored separately. The frequent k-itemsets are not included in further computation. The set of infrequent k-itemsets is used for the divisions. Once some data records have been replaced with the unions of their subsets and some are deleted, k increments and the modified dataset is again used for identification of k-itemsets. The algorithm terminates when no more frequent k-itemsets are found in data.

The frequent and infrequent itemsets are stored in hash tables. As k increments, the number of records decreases, therefore, the less computation is needed for finding k-itemsets in the next step.
Here, the division is performed on a set of m elements. If the m-itemset o contains one infrequent k-itemset, it can be substituted with k projections s_i of size *m-1* such that

$$o = \bigcup_{i=1}^{k} {s_i}^k$$

In other words, o is mapped into the set $\{{s_1}^k, {s_2}^k, ..., {s_k}^k\}$. If *k=2*, o is always mapped into a union of the two subsets s_1^2 and s_2^2 of size *m-1* each. It is obvious why — s_1^2 will not contain one of the infrequent items and s_2^2 will not contain the other. For example, if an infrequent *2*-itemset is $\{i_p, i_q\}$, then $s_1^2 = \{i_1,...,i_{p-1},i_{p+1},...,i_m\}$ and $s_2^2 = \{i_1,...,i_{q-1},i_{q+1},...,i_m\}$.
When *k=3*, an infrequent *3*-itemset, say $\{i_p, i_q, i_r\}$, projects o into
$s_1^3 = \{i_1,...,i_{p-1},i_{p+1},...,i_m\}$,
$s_2^3 = \{i_1,...,i_{q-1},i_{q+1},...,i_m\}$ and
$s_3^3 = \{i_1,...,i_{r-1},i_{r+1},...,i_m\}$.
It follows from the *apriori* property introduced in [16] that no superset of an infrequent itemset can be frequent. Then suffice to substitute o by the union of its *3* subsets of size *m-1* such that the first projection will not contain i_p, the second projection will not contain i_q and the third projection will not contain i_r. The ordering

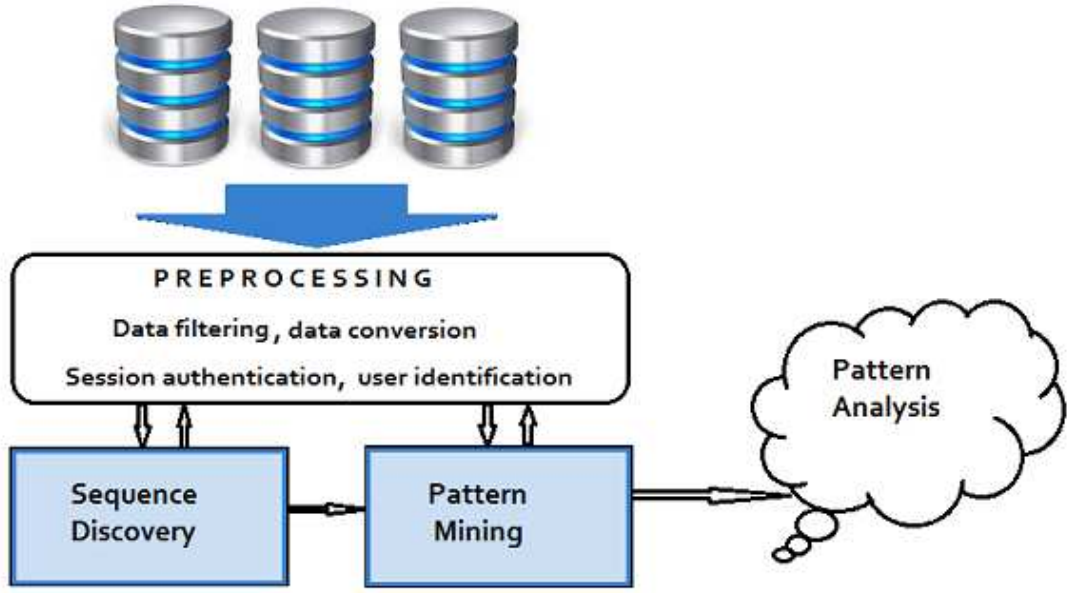

Fig. 2 Process of Web Usage Mining

within the union is not important, it is only required to maintain L-order within the itemsets.
When k=K, an infrequent K-itemset, say $\{i_1, i_2, ..., i_K\}$, is broken o in the same manner, that is, into the K projections $s_1^K = \{i_2, i_3, ..., i_K\}$, $s_2^K = \{i_1, i_3, ..., i_K\}$, etc, and $s_K^K = \{i_1, i_2, ..., i_{K-1}\}$. At each step k, the projections s^k should be of size k+1 or greater.

The first infrequent k-itemset used in division maps an m-itemset onto k subsets of size m-1. The projected once object o can be further projected with another infrequent k-itemset, in which case o will be mapped onto $L \leq \binom{m}{k+1}$ projections such that

$$o = \bigcup_{i=1}^{L} {s_i}^k$$

The value of L depends on the items contained in the k-itemset. For example, if the first k-itemset contains the item i_p, then as a result, at least one of the projections, say s, will not contain this item, e.g., $s = \{i_1, ..., i_{p-1}, i_{p+1}, ...i_m\}$. If the second k-itemset does contain i_p, s can not be used in breaking by the k-itemsets containing i_p.
If the second k-itemset does not contain any of the items of the first k-itemset, then L will depend only on the difference between m and k. If m-k is not big, then some of the projections may not be of the required size and, therefore, should be deleted. If one of the projections is a subset of another projection in a given union of subsets, it is deleted. All these factors are in favour of L be reduced.

If o contains p infrequent k-itemsets, then after p consecutive divisions o is finally mapped into the set of projections and is as follows:

$$o = \bigcup_{i=1}^{L^p} {s_i}^k$$

where $L^p \leq \binom{m}{k+1}$.

4.2 RecB: Computational Process

In the *RecB* algorithm, the process of projecting an object of size m by the infrequent k-itemsets contained in it is continued until it terminates when either:

(1) all projections of an object are reduced down to the length k+1; or

(2) there are no more infrequent k-itemsets left to continue.

With each infrequent k-itemset used for dividing an object, the first projection is obtained , then the second, the third and so on, finally the P-th degree projections, where P is the number of infrequent itemsets contained in the object .
The cardinality of a projected object is bounded by the sum of cardinalities of its L projections:

$$|\bigcup_{i=1}^{L} s_i| \leq \sum_{i=1}^{L} |s_i|$$

which follows directly from the *inclusion-exclusion principle* for sets.

The Erdös-Ko-Rado theorem introduced in [17] states that if $m \geq 2k$, then the maximal number of subsets of o of size k, each pair of which contains at least one common element, is $\binom{m-1}{k-1}$. The current example is less specific. The object o is of size greater than k. The subsets of o resulted from divisions may not be of the same size and may not have at least one element intersecting. In this case, a limit for the subsets of o is set to be of size k+1 or greater. Consequently, the number of projections of size no less than k+1 may vary from 2 (with projections of size m-1) to $\binom{m}{k+1}$ (with projections of size k+1).

The following is a pseudocode of the *RecB* algorithm, semantically based on the scripting language *Python*. Here, D is a dataset, k is the length of an itemset and σ is its support. The following data structures are generated during the computational process: *setOfAllKsets* is a hash table containing all itemsets of size k; *freqKsets, infreqKsets* - the two separate data structures containing only frequent and only infrequent itemsets of size k, respectively; *k-set* is a set of k items; *setOfCat* (a set of categories) is a hash table containing only frequent itemsets and is subject to pruning; *record* is a set of items, or a record in the dataset D; *infreqKsetsInRecord* is a list of infrequent itemsets of size k contained in *record*; *setOfProj* (a set of projections) is a set of subsets of *record* resulted after breaking a record.

RecB(D)

```
1   setOfCat ← { }
2   k ← 2
3   repeat:
4     freqKsets, infreqKsets ← { }
5     setOfAllKsets ← generateKsets(D, k)
6     for k-set in setOfAllKsets:
7       if σ (k-itemset) < minsup then:
8           infreqKsets.update[k-set]
9       else:
10          freqKsets.update[k-set]
11    for record in D:
12      setOfKsets ← generateKsets(record, k)
13      infreqKsetsInRecord ← { }
14      for k-set in setOfKsets:
15        if k-set in infreqKsets then:
16          infreqKsetsInRecord.update(k-set)
17      setOfProj ← record
18      for k-set in infreqKsetsInRecord:
19        setOfProj ← breakRecord(setOfProj,k-set)
20        for projection in setOfProj:
21          if len(projection) ≤ k then:
22            setOfProj.delete(projection)
23      if setOfProj = ∅ then:
24        D.delete(record)
25      else:
26        setOfProj ← removeSubset(setOfProj)
27        record ← setOfProj
28    setOfCat.update(freqKsets)
29    setOfCat ← removeSubset(setOfCat)
30    k ← k + 1
31  until freqKsets = ∅
32  return (setOfCat)
```

The function *generateKsets* has two input parameters. The first parameter is the original dataset D stored as a dictionary. The second parameter is the current value of k. The function returns the collection of k-itemsets in D - the hash table *setOfAllKsets*.

The function *breakRecord* has two input parameters: *setOfProj* and *k-set*. *setOfProj* is first initialised by *record* - a set of items, or a record of the dataset D. In the consecutive calls, *breakRecord* takes as an input *setOfProj* resulted from the previous division. *k-set* is an infrequent itemset of size k, by which divisions are performed, provided that *k-set* is contained in *record*. The function generates k copies of *record* (or a subset of *setOfProj*) and then removes one item —an element of *k-set* —from each copy. For example, if k-itemset is *A,B,C*, *record* is replicated three times with each replica missing either *A* or *B*, or *C*. An example further on in the section illustrates how *breakRecord* works. The function *breakRecord* returns *setOfProj*.

The function *removeSubset* has one input parameter - a set of itemsets. This function deletes the itemsets which are subsets of the other itemsets in the input set. The function *removeSubset* returns a set of supersets of the unput set. Note, when *removeSubset* is used for pruning *setOfCat*, it takes as an input parameter only a set of keys of the hash table *setOfCat*, as a list.

5. PERFORMANCE COMPARISON: THEORETICAL ESTIMATES

While most of the publications on pattern mining illustrate performance comparison for different algorithms on small datasets, our aim is to predict their performance on big data. This is usually achieved by obtaining the theoretical estimates of their complexity for the worst case scenario. The reasons for looking at the worst case are:

(1) It is useful to know the maximum amount of time taken on any input of size n.
(2) Average-case analysis is based on the knowledge of probability distributions of the input data and does require much effort to do.

This paper considers the base tree growing techniques because they have been recommended as web usage mining applications in [4][6][11][12][14][15][17]. The techniques introduced in the above publications are based on a tree building methodology, in particular, where the original web access sequence database is stored on a prefix tree (Tree). The algorithms mine the frequent sequences from the Tree by recursively re-constructing intermediate trees, traversing from suffix sequences to prefix sequences. The ways the Tree is mined can vary from algorithm to algorithm, but their core computational procedures are quite common. Let us estimate their complexity for the worst case scenario.

Keeping old notations, the input is a sequence of objects (records) $\mathscr{D} = (o_1, ..., o_n)$, where n is the number of records. Each o_k is a subset of the set $\mathscr{I} = \{i_1, ..., i_m\}$, where m is the number of distinct items (attributes) in $\mathscr{D}$. A pattern I is a subset of $\mathscr{I}$.

The most time consuming computational procedures in the tree growing agorithms are as follows:

(1) count supports for all items $i \in \mathscr{D}$
(2) identify all records o such that $i_j \in o$, $j = 1, ..., m$
(3) count supports for all $\{i_j | i_j \in o,\ j = 1, ..., m$
(4) compute possible combinations of $i_1, ..., i_m$ for each $\{I | I \in Tree\}$ and their supports.

The access time for each step will be denoted by τ, the average size of o_k by l, and the number of frequent items m_F. In our estimation, the procedures above will require the following complexity:

(1) $nl\tau$
(2) $mn\tau$
(3) $m_F nl\tau$
(4) $2^{m_F} m_F n\tau + 2^{m_F} m_F n\tau$

which in total, if bounded from above, is as follows

$$nl\tau + mn\tau + m_F nl\tau_3 + 2^{m_F} m_F n\tau + 2^{m_F} m_F n\tau$$

The following generalises that for the tree growing techniques, their performance is influenced mainly by n, l and m_F.

The local variables such as access time τ may be omitted thus the final estimate is as follows:

$$S_{FP} \approx nl + 2^{m_F} m_F n \approx 2^{m_F} m_F n$$

which puts the tree growing algorithms in the class of $O(2^m n)$ complexity, in the worst case. Here, m is the number of attributes and n is the number of records.

These are the *main memory* based mining techniques, and their cost in space consumption may reach $O(2^m)$. Each path in the grown tree will be at least partially traversed the number of items existing in that tree path (n in the worst case) times the number of items in the header of the tree (m). Therefore, the upper bound on complexity of searching through all paths will be $O(m^2 n)$. In order to establish which itemsets qualify to be frequent and keep all paths containing header-items, a tree growing algorithm may reach the complexity of $O(2^m n)$ order in the final stage.

Test runs on benchmark (small sized) datasets referred to in the works [4][6][7] [11][12][17][15][19] could not really show a marked slowdown on such a scale. Thus more testing would be desirable to demonstrate the algorithms' capability. It would be worth mentioning that in some techniques, recursions in counting create an additional computational barrier, timewise, which was acknowledged by some authors, for example, in [15][17][20][21]. Overall, the computational complexity estimate for the tree based mining techniques in question is exponential, in the worst case. In contrast, the *RecB* in the worst case will keep below or at the level of $O(KLl^k \bar{n})$.

While the theoretical estimates of computational complexity give us generalised answers, some experts still prefer to illustrate performance comparison with experimental results. The difficulty in obtaining such a comparison in our case is the unavailability of large sized benchmark web access data and the actual algorithms implemented in the same scripting language.

6. CONCLUSIONS

The problem of mining large sized logfiles in order to identify user activity patterns remains pervasive across many websites. The purpose of this paper is to advocate the development of highly scalable techniques for Web usage mining.
The proposed *RecB* algorithm provides the means to extract distinct features from mixed digital-text data structures such as web access logs. *RecB* is based on a very different principle compared to the algorithms described in [4][7][11][12][14] [15][17][19]. In particular, it does not build new data structures such as trees to aid pattern mining. Instead, it re-arranges the input data, which helps to save a great deal of computation later on.

This paper argues that however efficient a computational technique may appear on small datasets, it may not be as efficient on large datasets. If the performance of an algorithm depends more on the number of attributes than it depends on the number of records, test runs on small datasets may not capture that.

It also turns out that the number of scans through the data does not define how efficient an algorithm will be. Thus it can be rather limiting for pattern mining techniques to use newly generated data structures (candidate itemsets, trees) to optimise pattern mining. On large data, building numerous trees or generating itemsets requires much counting, so optimisation may become rather costly.[20][21] In contrast, *RecB* performs fewer scans across the data and converges in polynomial time. It avoids the pattern growth induced complexities by just re-shaping the original data, which helps to keep the computation within the acceptable timeframe.

The data cleaning and preproscessing stage is crucial for pattern mining in logfiles. Due to various challenges in identifying a unique user search request the suggestion is to apply as many domain specific filters to original records as possible and perform query string analysis to eliminate redundancy. Web log preprocessing techniques could also be applied to the area of intrusion detection in networks.

7. ACKNOWLEDGMENTS

We would like to thank our colleagues Graham Williams, Felix Andrews and Steve Curran for their generous technical support.

8. REFERENCES

[1] F.Giroire, J.Chandrashekar, G.Iannaccone, K.Papagiannaki, E.Schooler & N.Taft, *The Cubicle Vs. The Coffee Shop: Behavioral Modes in Enterprise End-Users* in: Passive and Active Network Measurement, LNCS 4979 (2008) p. 202.

[2] K.T.Kishore, S.T.Vardhan & L.N.Narayana, *Probabilistic Semantic Web Mining Using Artificial Neural Analysis*, International Journal of Computer Science and Information Security (IJCSIS) 7 (3) 2010.

[3] R.Cooley, B.Mobasher & J.Srivastava, (1999), *Data Preparation for Mining World Wide Web Browsing Patterns*, Knowledge and Information Systems 1 (1) 1999.

[4] R.Shettar, *Sequential Pattern Mining from Web Log Data*, International Journal of Engineering Science & Advanced Technology (IJESAT), 2 (2) 2012.

[5] V.Ciesielski & A.Lalani, *Data Mining of Web Access Logs From an Academic Web Site*, in: Proceedings of the Third International Conference on Hybrid Intelligent Systems HIS'03: Design and Application of Hybrid Intelligent Systems, IOS Press, 2003.

[6] Q.Yang, Ch.Ling & J.Gao, *Mining Web Logs for Actionable Knowledge*, in: Intelligent Technologies for Information Analysis, Springer-Verlag, 2004.

[7] R.Iváncsy, I.Vajk, *Frequent Pattern Mining in Web Log Data*, Acta Polytechnica Hungarica, 3 (1) 2006.

[8] L.K.J.Grace, V.Maheswari, D.Nagamalai, *Analysis of Web Logs and Web User in Web Mining*, International Journal of Network Security & Its Applications (IJNSA) 3 (1) 2011.

[9] O.Bell, M.Allman & B.Kuperman, *On Browser-Level Event Logging*, TR-12-001, ICSI, 2012.

[10] T.Callahan, M.Allman & V.Paxson, *A Longitudinal View of HTTP Traffic*, in: Passive and Active Measurement, LNCS 6032, 2010.

[11] J.Pei, J.Han, B.Mortazavi-asl & H.Zhu, *Mining Access Patterns Efficiently from Web Logs*, in: Proceedings of the 4th Pacific-Asia Conference on Knowledge Discovery and Data Mining, Current Issues and New Applications (PADKK'00), Springer-Verlag London, UK, 2000.

[12] L.Sun, X.Zhang, *Efficient Frequent Pattern Mining on Web Log Data*, University of Melbourne, Australia, 2004.

[13] L.Liu & J.Liu, *Mining Web Log Sequential Patterns with Layer Coded Breadth-First Linked WAP-Tree*, in: Proceedings of the IEEE International Conference on Information Science and Management Engineering, 2010.

[14] J.D.Parmar & S.Garg, *Modified web access pattern (mWAP) approach for sequential pattern mining*, International Journal of Network Security & Its Applications (IJNSA) 3 (1) 2011.

[15] K.C.Srikantaiah, K.Krishna, N.K.R.Venugopal, L.M.Patnaik, *Bidirectional Growth Based Mining and Cyclic Behaviour Analysis of Web Sequential Patterns*, International Journal of Data Mining & Knowledge Management Process (IJDKP), 03 (2) 2013.

[16] J.Han & M.Kamber, *Data Mining: Concepts and Techniques*, Morgan Kaufmann, 2001.

[17] P.Erdös, C.Ko, R.Rado, *Intersection Theorems for Systems of Finite Sets*, Journal of Mathematics 2 (12) (Oxford, 1961) p. 313.

[18] A.Robertson, *Permutations Containing and Avoiding 123 and 132 patterns*, Discrete Mathematics and Theoretical Computer Sciences, 3 (1999) p. 151.

[19] A.Rajimol, G.Raju, *FOL-Mine —A More Efficient Method for Mining Web Access Pattern*, Advances in Computing and Communications Communications in Computer and Information Science, 191 (2011) p. 253.

[20] F.Schulz, *Trees with exponentially growing costs*, Information and Computation, 206 (2008) p. 569.

[21] Sh.Cong, *A Sampling-based Framework for Parallel Mining Frequent Patterns*, PhD Thesis, University of Illinois at Urbana-Champaign, 2006.