Abstract: The World Wide Web (or Internet) implies from few years different uses. Each cybernaut wants, on the same medium, to retrieve information, to make shopping, to manage his/her bank accounts, etc. Customers or users want more and more features from the web. In order to realize services with high quality, to evaluate them or simply to design well a website, we want to know how surf or browse a user. Web usage mining is the field that deals with this kind of problems: automatically extract information about users, and then build knowledge about web usage or learn user behavior. In this paper we present a first required step which is the data pre-processing, with a small discussion about different kind of data can be accessible from the internet. Also, we develop our own session retrieving process and we show that structural model as automata can be easily used in a short task of usage prediction. Then, we carry out a set of experiments which show how our pre-processing method can outperform results on artificial data.

Keywords: Web usage mining, data pre-processing, grammatical inference.

1. Introduction

The Internet becomes more and more complex every day; few years ago, we can assimilate it as a huge document composed of only textual information. Nowadays, the World Wide Web is becoming a complex structure with a lot of exchanges between users. In fact, a cybernaut wants to have a lot of different services and for each of them a good quality. This growing complexity implies more and more difficulties in both ways: creating and accessing web data. The first item – which is the main point in web site designing, network designing and monitoring – aims to make available information. To well access data can not be optimal if previous item is not well made: the complexity of the Internet must disappear for the simple user who wants to read information through a lot of different pages for example. Browser (software client) development is a part of this aspect of the web. Usage information can be used in all these fields. When preferences or habits of the user are known, you can help him/her more or less easily by changing structure of the web site to make the users navigation better, or adapting network (by adding machines, cache, proxies or compressing length of path for accessing to a page), and developing client (browser) adapted to the user. Information concerning
Cataldi & al., 1995, Fayyad & al., 1996) deal with these themes, but only a recent one (Karampatziakis, 2004) presents grammatical inference as usage learning method. We want to show how a good pre-processing step can make learning one better.

Our contribution is divided into two main parts: pre-processing data and machine learning. Firstly, we discuss about the kind of data which are available in the www context. We will see that we're only interested on log analysis for reasonable points of view. We describe in Sect.2.2, server log files and network architecture related problems such as missing values. We define proxy and cache ideas, and how they can be annoying. Furthermore we study how to extract users group of queries called sessions. Section 4 explains a new way of mining web site usage with grammatical inference. We present here grammar models and known learning methods and we use them to extract user behavior. Use of grammatical inference is a very new way for this kind of data, first we simplify the problem not to extract complete behavior but to predict the type of the next page which will be asked by the user. The last section deals with set of experiments: we show better performance in learning task when good pre-processing is used.

2. Different data types

In order to learn user profile or usage on a specific web site, we want to grab a lot of data concerning how the user navigates. Available data are heterogeneous: users let behind themselves a lot of traces, from server log files to oculometric tests. We can classify theses traces into two types:

- active ones, where user give his/her own information;
- passive ones, such as log files, where no user action is required.

Another way to obtain information about user is to question him/her directly from the site via a form, and analyze the results. We chose not to work with these data: our process would be most universal as possible regarding different web sites, so we can't question users about his/her navigation because forms are generally developed specifically for one site.

We can split into two sides the data that we want to get: client and server side. Server side only represents log files and any data recorded on the web server such as web site graph for example. The other side is composed of all the data which are not available from the server: a client program (JavaScript), the mouse gestures, where the user is looking at, etc.

\section{Client side}

This part groups all data which is not recorded on the web server. We discuss here about how to get information about users clients.

Up to now, only ways to grab easily information concerning users (client side) was to make some little programs (JavaScript) which can send to server every mouse gestures or keyboard entry that a user make. At this time, spyware, for example, and other bots are malicious codes that enter the client and send back any information without user authorization. Obviously, we don't want to get information without user agreement.

Cookies are small pieces of text sent to server every time the user make a request to it, server can control what it is written (soon sent to it) in this file. They are not programs, just a text file which represents state of communication between client and server. There is no way to extract information only with cookies. We will see afterwards, how to use them to follow a user on a site. Even if there are some results (Caelen, 2001) on electronic document evolution by oculometric data we want to mention that these data are not reasonable in our idea of general usage mining. To obtain about 10 pages navigation result, which is scatter plot where points represent where user is looking at during the test, we need from 10 to 15 persons who can strictly no move during 25 minutes. These data must be interesting only for very fine tuning in web site designing.

Some of the well-used web browsers – such as netscape, mozilla, opera – accept local programming via an API. For this reason, if the user wants to send information to the web site, specific application for any client can be developed to monitor even the mouse movements, clicks or the keyboard events. This kind of data needs very large amount of coding to extract very short pieces of information. Actually, about each version of every browser requires a complete recoding to have a client working. Always in order to have a browser independent process, we choose not to use all client side data; we focus only on log records.

\section{Server side: the log records}

HTTP protocol (W3C, 1999) defined general behavior of a communication between web client and server by a way of height (extendable) method for the requests. Only two of them, GET and HEAD are required in the HTTP specification. Actually, the HEAD method is the same as GET except that returns only headers in answer without content. Therefore, we consider, in the rest of this article, only GET records from the log files. RFC2616 which describes the HTTP protocol, mentions that communication is generally done over TCP/IP connections: this point (TCP/IP connection) allows server to record, for every request, the address of the machine which requests it. Figure 1 summarizes simple communication between client and
server with HTTP protocol. Actually, in a real world, communication task is a little harder: between client machine and server, they can find a lot of machines with fully functionalities such as cache or proxy (see Sect.3).

Since then, a server can record interesting attributes: W3C recommends using the common log file format to record required fields (Luotonen, 1995). Generally, a log record is composed of:

- **remotehost field**: the address of the machine which makes the request;
- **rfc931 field**: the remote login name of the user. Actually, in order not to reveal remote login, this field is almost always blank;
- **authuser field**: the username as which the user has authenticated himself. This field is only filled when an authentication process is used.
- **[date] field**: data and time when the sever receives the request;
- **"request" field**: the request line exactly as it came from the client;
- **status field**: a return code informing the client of the request processing status;
- **bytes field**: the content-length of the answer transferred.

Table 1 shows an example of a log record.

<table>
<thead>
<tr>
<th>16.3.1.40</th>
<th>GET <a href="http://agra21.emse.fr/">http://agra21.emse.fr/</a> HTTP/1.1</th>
<th>200 OK</th>
<th>195.83.83.142</th>
</tr>
</thead>
</table>

**Table 1. Log record example**


3. **Caching and proxy features**

Caching has been created in order to reduce network traffic. Its principle is very easy: one of the machines situated between client and server (even client itself), have large disk space (cache memory) where is recorded previous requests and answers from the server.

As a result if a user re-requests the same data from the server, the cache re-send the same answer without requesting the server. We show here a situation to explain this: consider a network $I$, where two users (represented by their machines $M_1$ and $M_2$) are working.

They request a web site pages $A$, $B$ from a server $S$. Process is described in Figure 3.

Numbers printed near requests represents their order: that is, firstly, user $M_1$ gets $A$ page, then user $M_2$ gets the same page, then $M_2$ gets $B$ page, and so on. This screenplay can be represented as in Table 2.

**Table 2. Logs recorded by server for all requests**

| $M_1$ | - - | [30/Oct/2001:20:14:27 +0100] | "GET /A HTTP/1.1" | 200 | 293 |
| $M_1$ | - - | [30/Oct/2001:20:15:01 +0100] | "GET /B HTTP/1.1" | 200 | 148 |
| $M_2$ | - - | [30/Oct/2001:20:15:10 +0100] | "GET /A HTTP/1.1" | 200 | 293 |
| $M_1$ | - - | [30/Oct/2001:20:16:58 +0100] | "GET /A HTTP/1.1" | 200 | 293 |

Consider now, a caching system composed of two disk spaces locally placed into the user client: when client re-request the same page, local cache deliver it without querying the server. The same scenario is represented as in Table 3.

**Table 3. Logs recorded by client for all requests**

| $M_1$ | - - | [195.83.83.142] | 200 OK |
| $M_2$ | - - | [195.83.83.112] | 200 OK |
Table 3. Logs recorded by server, with local cache spaces

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>-</td>
<td>[30/Oct/2001:20:14:27 +0100]</td>
<td>&quot;GET /A HTTP/1.1&quot; 200 293</td>
</tr>
</tbody>
</table>

Table 3 shows that the last two pages were not requested again, because cache could easily deliver them to the clients.

Table 4. Logs recorded by server, with proxy-cache (global)

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>

Firstly, we can note that in the proxy-cache context, only different pages requests are recorded in the server log file. Because of that, recorded pages don't represent well the navigation of the two users. We also note, in this context, that we lose the requests sources, only the proxy machine address is recorded. This latter point can be very annoying: when we try to learn a user behavior, we must know the source of any request.

Server logs represent the main source of information from the user, without any specific or client oriented process. In order to discover user behavior we must split log records files into user sessions: some problems such as cleaning (some logs represent robots or pre-fetch process which not give you any information about a user), formatting (in order to use them easily) and missing values discovering (due to the cache and proxy systems) are very important to learn powerful models.

4. Splitting logs into sessions

In order to deal with this problem, we define user navigation and session. User navigation is a strict subset of all pages viewed by the user, without any other information or pages. This subset can be divided into several parts which each of them represents a semantical association of pages for one and only one purpose. Actually, a user can visit a site some times a day, for different goals: for example, on a news site, user can research one piece of information and later want to find another kind of news. These two semantical groups of requests are two different sessions.

Log file includes lot of different hits from all clients: viewed pages (accepted request), error handling (rejected requests), and many other system messages. So, we must extract pertinent records from the log file: by filtering source address, status code, and any information in log record. Furthermore, a web site can be viewed as a graph with each page is represented by a node and hypertext link as an edge. For this reason, if we extract easily the web graph, we can retrieve in log files some missing values by detecting incorrect navigation. For example, in a simple web site (view as graph in Figure 4), we can't access C page directly from B page; if in log file we find A-B-C sequence from the "same user", we can suppose that user requested for page A, then B, A again (not recorded because of cache feature), and C. In order to split log files into sessions, we scan it and a large problem (due to proxy feature) is to link a request to a user. In other words, considering a new record two cases can occur:

- If source address is different from all other sessions source addresses, in this case we assume that there is a new user on the system and we start a new session for him/her.
- Else, we search the better session (regarding accessibility of the page from web graph) with the same source address.

Figure 4. Simple web graph

Missing values

As we saw in Sect. 3, some requests can be non-recorded by a server because of the proxy and cache features on the network linking clients and server. There are some methods to try to retrieve lost information, we describe them here. We suppose that we are in a real context, so we can't assume that there is no cache or proxy between clients and server.

Cache-Busting

The concept of this method is really simple: cache
is annoying for our purpose, so we disable all cache machines. From the HTTP protocol 1.0 version, W3C introduced keywords allowing disabling cache (cache-control, Pragma: no-cache, Expires: 0). These features asks to the cache machines for not keeping pages copy locally. This kind of artifact can’t be used in a real context: firstly, the method is equivalent to suppress all cache features between clients and a server, which can increase the network traffic; secondly, for network robustness reasons, most of the cache machines don’t accept cache-busting silently. In this state, we can’t use this solution; we can imagine processing answer differently according the type of the page requested, just ask for a cache-busting for typical pages:

- member of a cycle (probability that a user asks many times this page is more important);
- node of the graph with big degree (probability to reach this state is also more important).

As we don’t want to increase the traffic, we don’t choose to use this kind of method, another amelioration of this solution is to make cache-busting not on all data.

**Sampling**

Sampling consists of cache-busting method applied either on only some kind of request, or only during a specific time. In (Pitkow, 1997), this method has been tested on real data. This is based on statistic results; they try to discover population properties analyzing only a sample set. Thus, to infer missing values, they enable cache-busting only at specific times or only from specific user (both randomly chosen). For same reasons as above, we prefer just work on server data, without modifying network traffic at all.

**Log file and Graph Detection**

Considering a web graph, we can detect some missing values without any other cache mechanism. This method is just based on detection of inconsistency in log file records. Consider the graph G(S, E) where G denotes pages (states) set, and E the hyper-links (edges). Let (v, w) be two nodes, we denote v → w the existence of an hyper-link between v and w pages.

When we parse the log file data, more than one session can be opened at a time (interlaced sessions). We keep in memory, the history for each session: let kth opened session has hk as history and cache processing: actually, different scenarios can be represented in log files as the same sequence. This kind of ambiguity doesn’t allow us to match well the user sessions.

Of course, we can’t retrieve all information lost in cache processing: actually, different scenarios can be represented in log files as the same sequence. This kind of ambiguity doesn’t allow us to match well the user sessions.

In order to evaluate which quantity of lost data we can retrieve, we chose to make some set of experiments on artificial data. We work on the web graphs extracted from two web servers (Agora21, Eurise). In order to simplify the task, and to test only the missing value recovery, we assign to each page a color: actually, we divide into two groups (blue and black) the web sites’ pages. We generate some possible sequences of visited pages (in fact, their corresponding colors) without any cache feature available in order to make a full log file. Then, we simulate some cache or proxy-cache machines available in order to make a full log file. After that, we simulate some cache or proxy-cache machines available in order to make a full log file. Then, we simulate some cache or proxy-cache machines between (imaginary) clients and server; we obtain an other log file with blanks. We split the two log files into sessions, we learn a stochastic grammatical model for each file, and we evaluate the quality of model without and with recovered missing values. The works of these experiments are display in Figure 5 and Figure 6.

### Function 1. Which Session

```plaintext
function WhichSession()
    Data: H, Ky //page we want to class
    minbackLength = no,
    for each k ∈ openSessions do
        for 1 ≤ i ≤ |P| do
            if h(i) → h(0) then
                if minbackLength > i then
                    minbackLength = i;
                    mSession = k; break;
                endif
            endif
        endfor
    endfor
    if minbackLength < no then
        /*we found an inconsistency into log records */
        AddSequence(M长效机制.mSession, minbackLength);
        Stochastic(m长效机制.mSession, h(0),)
        else
            /*there’s no available page in history; add a new session */
            l = NewSession();
            Stochastic(l, h(0));
    endif
endfunction
```

5. **Stochastic automaton machine learning**

### 5.1. Definitions

**Definition: Distribution over strings**

Let Σ be a finite set of letter, {a, b, ...}. A string w is an element from Σ*. A probability function is a function such that:

\[ \forall w \in \Sigma^*, 0 \leq P(w) \leq 1 \quad \text{and} \quad \sum_{w \in \Sigma} P(w) = 1 \]

**Definition: Stochastic Deterministic Finite Automaton**

A SDFA is a tuple <Σ, Q, q0, δ, p, f> where Σ is a finite alphabet, Q a set of states, q0 the initial state, δ a transition function, p a probability function, and f the probability that a state is final.

This kind of objects is used to learn a stochastic model on sequences. In our context, each user session is rewritten into a sequence of page types. So we have data which can represent the behavior of users on the web site. We can’t obtain negative data: in exact
learning theory, negative data are required to lead the learning algorithm. In order to solve this problem without any negative data (and also to deal with noisy data), we have to learn stochastic models.

5.2. Learning step

In machine learning theory, one way to solve the problem of inducing an automaton from positive data only (in our case, negative data for web navigation has no sense) is to use state merging algorithm for stochastic deterministic automaton: Alergia (Carrasco & al., 1994), MDI (Thollard & al., 2000).

This kind of algorithms is often defined in this way:

Algorithm 1. State merging general algorithm

$$\begin{align*}
\text{Algorithm 1: State merging general algorithm} \\
\text{Input: a set of positive samples } E \\
\text{a precision parameter } \alpha \\
\text{Output: a SDFA} \\
\text{begin} \\
\text{A} = \text{opt}(E); \\
\text{while } (s,j) = \text{select-state}() \text{ do} \\
\text{if } \text{similar}(s,j, \alpha) \text{ then} \\
\text{A} = \text{mergeAndErase}(s,j); \\
\text{endif} \\
\text{end} \\
\end{align*}$$

5.3. Evaluation

The “perplexity” is the more often used method to evaluate quality of stochastic models regarding the dissimilarity between the model A and a sample set S.

Definition: Perplexity

We note \( PP(\mathcal{A} | \mathcal{S}) \) the perplexity between \( \mathcal{A} \) and \( \mathcal{S} \) such that:

$$PP(\mathcal{A} | \mathcal{S}) = 2^{-1} \sum_{i=1}^{\text{length}(\mathcal{S})} \log_{2}(P_{\mathcal{A}}(S_{i}))$$

This measure represents the real difference of entropy between two entities, \( \mathcal{A} \) model, and the sample set \( \mathcal{S} \). This is an estimation of the Kullback-Leibler divergence (Kullback, 1959) on \( \mathcal{S} \).

6. Results

6.1. Description

Extracting web graph

From both web sites, we extract a graph which represents the site. We have a copy of the web site we want to adapt. Our algorithm starts from a unique index page and from page to page (by following html links), we discover and record all the accessible site.

Table 5. Sizes of used web sites

<table>
<thead>
<tr>
<th></th>
<th>#pages</th>
<th># paths</th>
<th># reference sessions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eurise</td>
<td>105</td>
<td>1000</td>
<td>148</td>
</tr>
<tr>
<td>Agora</td>
<td>4850</td>
<td>1000</td>
<td>137</td>
</tr>
</tbody>
</table>

Generating paths

As we said before, cache and proxy features don’t allow having all the requests from the users. In order to show how our method works to discover missing values, we have to generate artificial data. We generate, for each web site, 1000 paths. Each path is associated to a target page, and is generated by computing the shortest path to access the target page, and by adding some randomly “human” behaviors: we allow to go back, and to go out the shortest path by choosing a wrong link in a selected page. Of course, this method introduces bias in the generated data, but it is a simple method in order to know if our process is useful or not.

Simulating cache and proxies

We want to show that we can do better learning with missing values retrieving so we have to test on complete data and on a subset. We developed a cache-proxy simulation, which clear logs that would not be requested in the real life according to a proxy-cache configuration. We assume that logs are requested from a set of machine, ones behind proxy, and all with a cache. After this step, we have to sets of data for learning and compare the models.

Learning

In order to evaluate learned models, we split each log files (complete data and cache-proxy modified one) into train and test sets. Train set is transform into “users” sessions to learn a model of browsing behaviors, test one is used to compute perplexity. To learn the automata, we use logs sessions rewritten into color sequences, such as ‘blue, blue, black’ for example. Algorithm used for learning is the well-known Alergia (Carrasco & al., 1994) which uses state merging method. The criterion used to merge
states is based on a statistical test (Hoeffding bound). The quality of models is evaluated by perplexity (Carrasco, 1997). Results are shown in Figure 5 and Figure 6.

In order to make more robust our results, we compute perplexity many times by changing train and test sets: for example, with a sets of 7000 sessions, experience with number 0 will be composed of a train set with the 1000 first sessions and test with the 6000 last ones and experience number 4 will have a train set with sessions from 5000 to 5999 and all others into the test sets. X axis displays the number of experience. Y axis represents the perplexity: the lower the perplexity is, the better the results are. The solid line shows the reference results on the artificial complete logs (in the real life, we can’t obtain these data). The dot line shows learning results on the logs with missing values. The last line (dashed) shows our method with missing value recovering.

Table 6. Perplexity mean on the three different methods: cache-busting, with real-life logs (missing values), recovering (our method)

<table>
<thead>
<tr>
<th></th>
<th>reference</th>
<th>missing values</th>
<th>recovering</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eurise</td>
<td>2.31</td>
<td>3.39</td>
<td>2.56</td>
</tr>
<tr>
<td>Agora</td>
<td>2.08</td>
<td>2.57</td>
<td>2.31</td>
</tr>
</tbody>
</table>

We can note that our method give really better results than the classical case, almost the same as on the real complete set of logs. Another important result: models learned without missing values recovering are really poor: actually, data are really confused, some sessions are mixed together and no clear behavior can be extract.

7. Conclusion and further work

We showed that with our method, when logs are pre-processing in order to recover missing values due to the presence of some network features, the results in learning step with grammatical inference are better.

We want to develop some ideas: firstly, we want to obtain robust results in the same way, not only on the simple task of extract behavior from a 2 letters alphabet (pages are colored bleu or black) but on all the pages of the site, in order to predict the exact next page. Another point will be to learn how proxy machines work directly from data: we get complete data logs by cache-busting for a very short time and we compare with data obtained after via proxies. With the same technique, we want to learn also dynamically, the graph structure of a web site from the logs.

8. References

Agora21, web site (http://www.agora21.org)


Eurise, web site (http://eurise.univ-st-etienne.fr)


