A Web Crawler Framework for Revenue Management

DANIEL MARTINS, ROBERTO LAM, JOÃO M.F. RODRIGUES, PEDRO J.S. CARDOSO
University of the Algarve
Instituto Superior de Engenharia, LARSyS
Campus da Penha, 8005-139 Faro, PORTUGAL
{djmartins,rlam,jrodrig,pcardoso}@ualg.pt

FRANCISCO SERRA
University of the Algarve
Escola Superior de Gestão, Hotelaria e Turismo
Campus da Penha, 8005-139 Faro, PORTUGAL
fserra@ualg.pt

Abstract: Smart Revenue Management (SRM) is a project which aims the development of smart automatic techniques for an efficient optimization of occupancy and rates of hotel accommodations, commonly referred to, as Revenue Management. To get the best revenues, the hotel managers must have access to actual and reliable information about the competitive set of the hotels they manage, in order to anticipate and influence consumer’s behavior and maximize revenue. One way to get some of the necessary information is to inspect the most popular booking and travel websites where hotels promote themselves and consumers make reservations and provide reviews about their experiences. This paper presents a web crawler framework to perform automatic extraction of information from those sites, to facilitate the (RM) process of a particular hotel. The crawler periodically accesses the targeted websites and extracts information about a set of features that characterize the hotels listed there. Additionally, we present the document-oriented database used to store the retrieved information and discuss the usefulness of this framework in the context of the SRM system.

Keywords: WebCrawler, NoSQL Databases, Revenue Management.

1 Introduction

The objective of Revenue Management is to establish strategies based on the understanding of market dynamics, to anticipate and influence consumer’s behavior and to maximize revenue and profits from a fixed resource. The amount of data required to produce optimal decisions is huge, justifying the adoption of the concept currently known as big data [1].

For a number of years, the hospitality industry and its partners, e.g., Global Distribution Systems (GDS) and Online Travel Agents (OTA), have been promoting their transactional services on the web [2]. These sites provided hotels with certain types of business information on a free partnership base, because they had a great need to promote their services, thus making them willing to promote alliances and to facilitate information, as a means to achieve high growth rates. This was typically a win-win situation but, more recently, this scenario is changing in what concerns hotel-OTAs business relations.

Due to the intense competition, hotel managers are trying to promote their services at OTAs’ sites that hold the highest market shares. These predominant OTAs, taking advantage of their privileged position, started to demand extra fees for promotion and for the facilitation of business intelligence data, besides higher booking commissions. As stated before, hotel managers strive to achieve the best possible revenues but in order to do that, they need to be in possession of actual and reliable information about their competitive set (e.g. hotels with similar location, facilities, class of service, number of rooms, Guest’s Reputation Index) and about the corresponding total demand.
One way to get some of the data is to inspect the sites where the hotels of the same competitive set are doing their promotion and bookings. The simplest form, although not the cheapest one, is to contract the access to business data through an API commercialized by the OTA of interest. More complex is to get the information from HTTP, simulating the behavior of a user (not the same as “hacking”).

The latter, extraction using a web robot (bot) or crawler, makes it possible to get partial data (e.g., prices, room types, capacity, facilities, amenities, as well as comments from former guests), available on the hotels. The amount of data to extract is huge, since the crawler must run periodically in order to extract updated data.

The OTAs’ sites, like Booking [2], protect their data from those extraction processes for two reasons: (a) They do not want to have bots scrapping their servers, since this will cause an overload of the systems and consequent delays in the normal processes of promoting and booking and; (b) They want to sell the access to the data through the API of their engine site. Nevertheless, research has been conducted on data extraction from web information systems [3, 4, 5, 6], but only a small number of studies have been published on the subject of business to consumer (B2C) [7].

Smart Revenue Management (SRM) is a project in development by the University of the Algarve and VISUALFORMA - Tecnologias de Informação, SA, which aims to develop a set of tools to integrate in a RM system. In this paper, we present some part of that tool, namely a web crawler framework for smart RM. The main contribution to that system is the smart web crawler and its associated database to store the extracted information.

2 Contextualization and State of the Art

The World Wide Web (WWW) is currently the largest source of information available to the public. The increasing use of the internet, worldwide, has made e-commerce to evolve, thus facilitating the emergence of an ever-growing number of sales channels and business opportunities [8].

Data extraction from the web, including from e-commerce sites, is useful for many types of business analysis, allowing the use of predictive models to enhance the performance of the revenue management system [9]. The extraction of huge amounts of data from the web requires almost impracticable time and effort for humans, thus justifying the creation of mechanisms for its automatic extraction [10].

When the HTML structure of the pages is constant, creating a mechanism that is able to automatically parse and extract data from a particular site is not a difficult task. However, if the site administrator/programmers decide to change the structure of the DOM tree or the attributes contained in the tags (e.g., class or id values), it is mandatory to (re)implement/adjust the extraction mechanisms (this is what happens in the websites used to extract the information for the RM).

Over the last decade, several studies have been conducted about the automatic extraction of information from the web. Lerman et al. [11] presented a fully automatic system that can extract lists’ and tables’ data from web sources through a series of assumptions about the structure of those lists and structures. In [12] it was used an algorithm that requires the user to identify the relevant data to extract. The algorithm uses the Minimum String Edit Distance procedure and is able to identify the tags that are normally used for that type of data. A system that compares two HTML pages to find patterns was proposed in [13]. It uses two web pages with similar structure but with different contents in order to identify patterns between the pages and to create extraction rules.

The Tree-Edit Distance algorithm was used in [14] to find patterns between different structures. Initially the HTML pages are converted into a DOM tree. The Tree-Edit Distance applied to two DOM trees computes the distance

---

1A DOM tree is a tree where the root is the <HTML>.
between two trees, $T_a$ and $T_b$, which is the cost associated with the minimum number of operations needed to convert the $T_a$ tree into the $T_b$ tree. These operations can be the insertion, the removal or the edition of nodes. Then the system does the page clustering which is the grouping of pages with a certain resemblance. Once the pages are grouped, the tree edit distance is used to generate the extraction pattern applied afterwards. A similar implementation, based on pattern analysis and through DOM trees using the edit distance algorithm, was used in [9].

The VINTs (Visual Information and Tag structure based wrapper generator) was proposed in [3] to extract data from search engines results. The VINTs refers to the visual content of the web page to find regularities / contents order, without being interested on the page’s HTML code. Those existing regularities are combined with HTML code regularities to generate the wrappers.

Papadakis et al. [10] presented a way to figure out the format of the information contained in web pages and discover the associated structure. This system consists of three distinct phases. In the first phase, the system transforms the HTML document into a XHTML document through some syntactic corrections, making it “well” structured, and generates the correspondent DOM tree. In the second phase, the regions containing information of interest are segmented. Finally, the third phase consists of mapping the nodes of interest in the original HTML page.

Zhai and Liu [15] presented a system that only requires a sample page labeling. They use a method called Sufficient Match to identify the similitude between the objective page and the main sample page. The ViDE (Vision-based Data Extractor) [16] is a recent method of data mining that relies on the visual aspect of how the data is presented. The ViDE is composed by ViDRE (Vision-Based Data Record Extractor) and ViDIE (Vision-Based Item Data Extractor). This system operates in four distinct steps: First, the visual representation of a sample page is computed, which is then transformed into a visual block tree. Second, the system extracts the data records contained in the visual block tree and the third consists in the separation, supported in semantics, of the extracted data in “data items” and groups. Finally, a set of visual extraction rules to produce the visual wrappers is generated.

However, the previous methods do not take into consideration a major problem, posed by the dynamic pages that use JavaScript. Many Web pages use JavaScript to trigger dynamic changes in the HTML code without any request or response from the web server. In particular, JavaScript is employed in many e-commerce sites to hide information and to difficult the automatic data extraction task. Since these scripts cause changes in the structure of the HTML code only in the client side, it is necessary to interact with the web page so that information becomes visible.

Baumgartner and Ledermüller [17] presented a method, which overcomes this problem. For that, they have proposed the Lixto Visual Wrapper, which integrates the Mozilla browser driver to interact with the web page in order to display the information that is hidden in the backend database. The Lixto Visual Wrapper allows the user to view the page, to extract the data from that page and to interact with it, by sending commands from the keyboard or mouse. Those keyboard and mouse commands are recorded to the element, as well as the XPath, (the path to a given node in a DOM tree) that is to be extracted.

The previous methods are, in general, specific to the data extraction from particular web pages, which are easily transposable to a database schema. For these reasons, most systems store data in XML or relational databases. However, our system aims to extract data from multiple e-commerce sites. The issue is that each vendor has different ways of describing the same product [9]. Therefore, it is almost impracticable to create a well-structured database schema for a set of e-commerce sites.

Thus, our framework uses MongoDB [18], a non-relational database. MongoDB is a NoSQL [19] document-oriented database, with
high performance and high reliability. Other characteristics include easy scalability (vertically and horizontally through replication and auto-sharding, respectively) and map-reduce.

A MongoDB database is structured as a set of collections, which store documents. These documents are BSON objects (binary JSON [20] document format), allowed to have a dynamic schema, i.e., documents in the same collection are not forced to have the same structure. This schema-less property is particularly important in the problem in study since the data retrieved from the sites does not follow, in general, a common design. Furthermore, in our case, the data sources in question are themselves quite dynamic, including and removing new fields very often, which makes it very hard to design and maintain a relational database schema.

3 Web crawler framework

As mentioned, the amount of data needed to feed the RM system is huge and the crawlers must run periodically in order to update it. Nevertheless not all the data that is extracted can be used, as is, by the RM models, and from different sites it is possible to extract different and coincident information from the same hotel.

Due to the above reasons, we choose to apply a crawler to each site (Booking, Expedia, TripAdvisor, etc.), extracting periodically all the information existing on the site over different periods of time, simulating different users (2 adults, 2 adults and 1 kid, etc.). In this paper, we will only address the crawler and a primary database, where the “raw” extracted data will be stored. Nevertheless, it is important to comprehend the context of the framework, namely that from this raw data, a secondary database is being created using rules and semantics to produce the necessary filtered data for the RM models. Figure 1 shows the resumed block diagram of the entire framework, where the two darker blocks (in blue) are the ones that we will address on this paper.

3.1 Implementation of the crawler

As stated in section 2, research has been done about methods to automatically extract data from web sites. However, only a few publications mentioned the specific dynamics of e-commerce pages and the ways to deal with them. Most of the e-commerce websites development use JavaScript and AJAX to implement the characteristics associated with e-commerce (e.g., motion, pre-filled data and suggestions). The e-commerce sites are designed for human interaction, disclosing information to the costumer according to the previous inputs. To overcome the complexity of the interaction with this kind of websites we decided to use a webdriver.

The W3C [21] defines the specification for the WebDriver API as a platform and language-neutral interface and associated wire protocol that allows programs or scripts to introspect into, and control the behavior of a web browser [22]. The WebDriver API is primarily intended to allow developers to write tests that automate a browser from a separate controlling process.

In the previous context, ChromeDriver [23] and C# allows simulating the human behavior in the interaction with the browsers/sites. Together they allow to playact the process of inserting text in entry fields and doing clicks on other input elements (e.g., checkboxes, radio-buttons, submit buttons).

We must now recall that the goal is to extract information from websites that show lists of hotels and allow the booking thereof. In general, these sites work in a similar manner: (1) their homepage have a form to allow searching for a type of hotel, city and a period of stay. After fulfilling the form, a (2) list of hotels that match the criteria will be returned by the site. Clicking on one of these listed hotels, (3) a new page will be showed, exposing information about the selected hotel to the user, namely: available rooms, prices, feature, amenities, policies, guest comments, etc.

It is in the interest of the SRM project to extract information about the list of available rooms and corresponding prices, as well as re-
views made by the former customer of the hotels. Almost all websites of this kind provide this data. The main obstacle in building an automatic mechanism for extracting data is the fact that each site provides this information in the manner that it considers most appropriate for a human customer. Beside this fact, due the market competition, there is no interest of these websites in granting free access to automatic extraction of the information. Some of those sites change, from time to time, their pages layout (design), as well as the attributes values of tags, which contain important information for our purposes.

Our algorithm of extraction is described in the following basic steps:

Step 1. Set the URL to the website to be crawled (e.g., Booking, Expedia); set the data to fill the website forms (e.g., location, check-in and check-out dates, number of persons, number and type of rooms available) and other parameterizations (e.g., language, currency);

Step 2. Automatically fill an instance of the webdriver which models the behavior of the website to be crawled (using data from Step 1), and do a request to the corresponding server;

Step 3. Store the response of the server request, into a list of links referencing hotels and boarding houses that satisfy the research domain of Step 1;

Step 4. For each link of Step 3, do a second level of request to the server and extract all relevant data (e.g., type of hotel, available rooms, prices, comments and rank position);

Step 5. Stored the retrieved data on MongoDB database.

The process of finding the important HTML elements (those containing pertinent data for RM), will be described in following sections.

3.1.1 Relevant HTML elements

Despite the use of Javascript and AJAX at the front end of the system, it will be a browser rendering the final contents in HTML [21]. Due its intrinsic markup nature, HTML is a language that allows the construction of blocks of nested tags. Thus, in order to extract the value of a specific element with a specific tag, we can use the XPATH definition to do this [21]. The alternative to identify the relevant elements is to locate the attribute associated with a tag.

For instance, the tag <div class="hotels" id="Hotel1"> </div> contains two attributes; class and id. Their values are “hotels” and “Hotel1”. The attributes are defined by pairs of name = value. The use of the absolute XPATH for accessing a particular element has a main drawback: if there is any change in the page structure the tag might no longer be accessible. To cope with this inconvenience we first use the attributes. If the tag element does not have an attribute, then the relative XPATH will be used. To surpass the problem of absolute XPATH we use a tag from higher level and, from there, the relative XPATH to the target tag is used.

In Fig. 2 middle, if we would try to extract the text “We got upgraded to a ...” it is necessary to use the relative XPATH, because the text is inside a <span> tag without attributes. The relative XPATH to the text would be //div[@class='reviews-carousel-scroll'] /div/p/span, which means: find the tag <div> with the attribute class equal to
reviews-carousel-scroll and from it follow the /div/p/span relative XPATH to the target tag.

The information about the target tags for each website are manually provided and placed in a database. The webmasters of e-commerce websites tend to change the attributes values periodically, although not so frequently, to avoid user/client annoyance. Besides, complete renewals of the website take place from time to time. In this case, there is no other solution than to redo / redefine the target tags.

Nevertheless, for the first case, where the attributes are changed, we use a vector for each target tag, where all the previous used tags are kept, and register how many times these tags has been used. When one of the tags is not found, the crawler goes to the correspondent storage vector and try to find a correspondent tag (from those already used and stored in the vector). Usually, the web programmers employ the change of attributes values in order to block the automatic data extraction mechanisms but as humans, they start reusing values. If the attribute value has never been used a counter is used in order to keep the occurred number of failures, this will allow determining the ratio of unsupervised versus supervised data extraction data.

### 3.1.2 Data extraction from HTML tags

After the recognition process, it is necessary to extract the information from the HTML tag. The information to extract is located between the begin and end tags of the element (e.g. `<div id="hotel_name"> Hotel </div>`). However, some of the information is not showed as explicit text. For instance, some websites show the number of stars of the hotel as an image (Fig. 2). In this case we extract a relevant attribute value. The images used in e-commerce website have attributes to help the posing the image in the page and defining the semantic importance of the image, see Fig. 2.

### 3.2 Database implementation

Taking into consideration that distinct websites have distinct designs and structures, we have chosen MongoDB (see section 2). MongoDB allows to storage data in collections. In our database there are four collections: AboutHotel, Rooms, Comments and Scores.

All the data concerning the hotel characteristics (e.g., facilities, name, address, and number of star), are kept in the AboutHotel collection. Figure 3 shows an example of the data extracted from TripAdvisor for a 4 star hotel (in this figure and similar ones, some fields were truncated for questions of size). Data for the same hotel, extracted from Booking is presented in Fig. 4. As we can observe, there are significant differences between sites both on type of information and respective details.

The Rooms collection keeps the information about the rooms of the hotels (e.g., room name, type, capacity, prices/dates, and used taxes). Figure 5 presents an example of the data retrieved from Booking. The Comments collection keeps comments made by formers guests. In this case, the comments are grouped by the
type of guest (e.g., couple, family and friends) – see an example on Fig. 6. The Score collection keeps the scores given by former guests for evaluations of cleanliness, staff kindness/efficiency and comfort of the hotel – see an example on Fig. 7.

Each of these collections has a general structure capable of storing data from this type of websites described above. By general structure, we mean the subject lists and tables of data for each website, kept with no dependencies in name or content manner; see an example on Fig. 8.

In order to keep the independence from the
"title" will possess the name of the subject in the layout of website and the "content" will be the content of that subject.

The proposed database is the primary database, with the propose of retrieving and storing has much data as possible from the websites. Later, in dependence of the aim/purpose of research, a large number of studies can be conducted. Data can be grabbed from the database and clustered in a single unit of information or, it can be used to create time series, in order to analyze past performance or to predict future trends (see Fig. 1).

4 Discussion

The usefulness of the RM system is based on the availability of data. In order to satisfy this
need, the crawlers must run periodically in order to collect it in a suitable and updated manner. If this procedure is repeated with short intervals, the hoteliers can get valuable information to organize data series that can be used with predictive algorithms to decide on the best prices and service-mix strategy, in order to obtain higher revenues. In this paper, a web crawler framework for RM was presented, aiming to demonstrate that it is possible to automatically “browse” e-commerce websites, identify the relevant elements and extract them to a NoSQL database.

We were able to overcome the interaction of JavaScript and AJAX by using a webdriver and, although the extraction of data can not be a fully unsupervised process, human supervision is only required if page layouts are modified. In the future we pretend to improve the web crawler, so it can detect and “understand” layout modifications, adapt to them without the need of human supervision, provide metrics on the history scheme for the replacement of the attribute values (tags), and register the correspondent metrics in the database. All these metrics will be computed when the full integration of the system occurs, i.e., when the primary database (presented in this paper) is fully integrated with the secondary databases (out of the focus of this paper), which integrate the information extracted from the web crawlers of the different sites and provide formatted data for business intelligence purposes. These secondary database will be the one where the mathematical models of the RM will be based.

Acknowledgements: This work was supported by FCT project LARSyS (PEst-OE/EEI/LA0009/2013), CIAC (PEstOE/EAT/UI4019/2013) and project SRM QREN I&D, n. 38962. We also thanks to project leader VisualForma - Tecnologias de Informação S.A.

References:


