POLYTECHNIC OF TURIN

Master of Science in Computer Engineering

Master's degree thesis

Design and implementation of tools for automatic detection of web page rendering problems

Advisor

Prof. Marco Mellia

Co-Advisor:

Dott. Ing. Martino Trevisan

Candidate

Michelangelo Sorice

Company tutor

Stefano Traverso, PhD
Ermes Cyber Security

Academic year 2018 – 2019

Summary

In the context of modern Internet, different actors can be identified. End users access and enjoy contents published by websites, on their side websites need to monetize their business and to do this they generally rely on advertisement. Nowadays advertising has become a smart and data-driven task, to increase adverts effectiveness, specific ads are targeted to specific user according to their preferences. This is possible thanks to other actors called web trackers which, exploiting constantly evolving techniques, are able to collect a wide range of user's personal data and exploit it to profile their tastes with threats for privacy and security.

Plenty of tools have been developed aimed to mitigate the threat represented by trackers, yet the protection provided by these tools is in most of cases actuated by blocking traffic towards and from tracking services which, in many cases, are also content providers hosting web resources that are crucial for the correct rendering of websites relying on them.

For this we need to refine tracker blocking tools in order to **make them aware of eventual problems they could cause to the rendered page**. Understanding when a page is not properly rendered is not a simple task: first we need to identify which parts of pages that nowadays are full of advertising and noisy contents, actually need to be preserved, in other words we have to identify sections which contain core contents. Knowing that, we would be able to refine the rules driving tracker blockers in order to preserve user privacy without compromising page functionalities.

The **aim of this research work has been developing a solution enabling the automatic recognition of core sections of web pages** thus allowing automatic detection of eventual rendering problems. This problem has some points in common with the web content extraction problem which has been subject of several previous studies but, as far as we know, no one had targeted this specific aspect yet.

The proposed solution is **based on the assumption that core and functional parts of the page tend to be less mutable with respect to less relevant ones** especially considering relatively close in time visits. From this starting point, we developed a visual approach to the problem based on the comparison of screenshots collected during multiple, close in time visits to the same page aimed

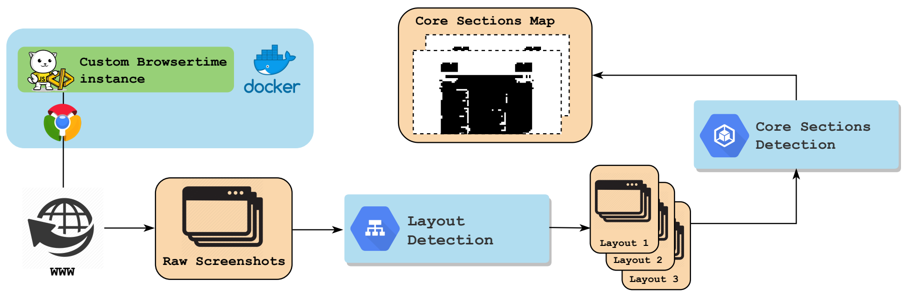


Figure 1: Schema representing the system pipeline.

at discerning static (and thus more relevant) sections from more mutable ones. To the limit of our knowledge this is an innovative approach even in the context of web content extraction.

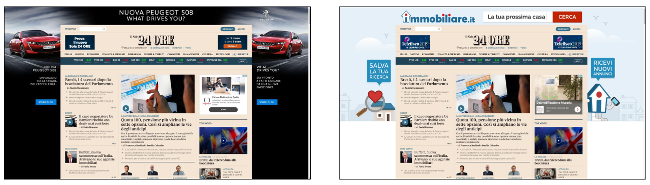
The developed system pipe, presented in Figure 1, has three stages: **data collection**, **layout detection** and **core sections detection**. During the data collection phase, we perform multiple visits to a certain website collecting captures of the rendered pages within a limited amount of time. This operation is performed using Browsertime [1] which is a flexible, open source tool able to instantiate a web browser and perform automatic visits to a website collecting several types of metrics, including page captures.

To evaluate the mutability of page contents it is crucial to **compare captures showing the same page layout, meaning the same spatial disposition and shape of static and dynamic contents**. Unfortunately, nowadays even simple websites tend to render the same page with multiple layouts usually differing in the amount, disposition and shape of dynamic contents. The second stage consists in processing our screenshots by mean of an algorithm able to detect the different page layouts being rendered by a website and group captures implementing the same one.

The layout detection algorithm compares captures attempting to state if they implement the same layout or not. Comparisons are performed splitting both captures into a grid of identical sub blocks, then we compare corresponding ones and create a *changesMap* keeping track of which blocks differ among the considered captures. This map is evaluated using three routines: the first one computes the percentage of differing blocks, under a certain threshold we assert that captures implement the same layout because of their strong similarity. The second one performs the same kind of evaluation but limited to blocks laying within the core part of the capture. We perform a crop, defined by the algorithm parameters, of the central section of the capture: in the vast majority of websites this section contains less mutable contents in which differences due to differing layouts are easier to spot. Thanks to this second routine we are able to detect that captures in Figure 3.5 belong to the same template despite the noise produced by the differing

[1]For further details https://www.sitespeed.io/documentation/browsertime/

background banners, Figure 2b shows a visual representation of the *changesMap* derived from captures comparison with differing blocks filled in black and the core area highlighted in green. The third routine is based on a different intuition: in most cases the difference among two layouts is the presence of an additional element (usually a banner), the effect of this addition is always that of moving all the static contents under the new banner, this change results in particular patterns on the *changesMap* which can be exploited in the layout detection task. Thanks to this qualitative rather than quantitative measure of layout similarity, we are able to group captures even in extremely noisy pages as those showed in Figure 3 in which almost all the page consists of dynamic and thus non relevant contents.


(a) Evaluated captures.

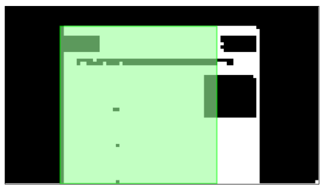

(b) Derived *changesMap*.

Figure 2: Captures grouped thanks to core level similarity evaluation.

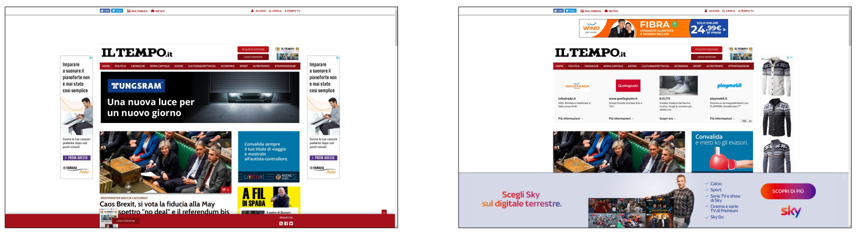


Figure 3: Captures grouped exploiting *changesMap* patterns evaluations.

Once we obtain a set of captures implementing the same layout, the core sections detection algorithm compares them to identify the mutable parts of the page. The comparison operations are similar to those performed for the layout detection stage but we use a more fine grained grid. After performing a full mesh of comparisons we obtain a set of *changesMap*, which are evaluated to take a final decision over the nature (static or mutable) of each block. The final output is an accurate, punctual map of core and mutable sections of the page.

The algorithms were tested using different datasets. For layout detection we considered a set of ten websites differing for the complexity and the amount of rendered layouts, one hundred captures were collected for each test case. For this dataset we built the ground truth by hand, providing a file containing the optimal grouping for each test case. For core sections detection instead we produced a synthetic dataset: we implemented an angular application able of generating several page layouts composed of dynamic sections (whose content could randomly change at any visit) and static ones. Given a page configuration the page generator produces also the corresponding optimal *changesMap* which is used as ground truth. Five page layouts were defined inspired to common modern pages layouts, we built our synthetic dataset by collecting one hundred captures for each of them.

The complexity of our one of a kind problem implied the need of designing and implementing from scratches an innovative solution along with a whole testing system and proper datasets, it required three months for the implementation. We spent over **150 hours testing and refining the configuration parameters** on both algorithms using a **low budget hardware configuration**. For the layout detection algorithm, we evaluated over **400 possible configurations**, setting step by step its **17 parameters** in order to find the optimal trade off among accuracy and time performances. The best performing configuration achieved the following results: on average the **98.6%** of captures is assigned to the correct layout and the **95.1%** of layouts is identified, in the **80% of test cases the algorithm performed a perfect detection**[2]. The average run of the algorithm on a set of one hundred captures lasts **40.6 seconds**. We obtained equally outstanding performances in core sections detection achieving the **100% of accuracy** in all tests cases, we found that the same accuracy could be achieved limiting the amount of captures evaluated for each test case down to a minimum set of 30 units being able to perform a perfect core sections detection in **63.2 seconds**.

During the research work, the tackled problem revealed its complexity due both to the logical complexity of deciding when the functionality of a page is compromised and to the inherent complexity and variety of modern websites. We were not able to complete a system for the automatic refining of tracker blocking tools rules, yet **the developed solution provides a fast, accurate and above all general purpose way to identify the actually relevant parts of a website**, enabling future studies to measure the effects on the rendering of these contents of tracker-blocking tools. Moreover the visual approach adopted introduces an innovative paradigm that, in future studies, could be further explored by content extraction techniques. The developed system itself has margins for improvements: larger and more complex datasets as well as more extensive tests could allow further refining of parameters and prepare the system for use in the wild web.

[2]100% of captures correctly classified and all templates identified.

Contents

Chapter 1

Introduction

Nowadays web surfing and online interactions represent a consistent part of our daily activities. We are constantly connected, and we are constantly exchanging data. Despite this deep interactions we are not completely conscious neither of the real volume of data that we share nor of the nature of these informations. Being our lives so strictly influenced by the Internet, awareness of how it works, that is how data are exchanged and how they can be used to generate profit, is crucial both for people and companies in order to not underestimate possible threats and avoid the unwilling disclosure of sensitive informations to third parties.

1.1 The Internet Ecosystem

Within the modern web ecosystem we can identify four main kinds of actors. First we have *websites* which in many cases offer and eventually produce contents. These contents are, in the vast majority of cases, freely available to end users and that is the reason of the great success of Internet. A relatively unlimited amount of news, knowledge, entertainment and literally any kind of content which can be conveyed through the Internet media is available for free 24/7.

Of course content producers and deliverer must have an economic return from their work in order to build a sustainable business on it. Ever since the first years of widespread adoption of Internet, the vast majority of services and websites on it have based their business model on *advertising*, this is no wonder if we think to the unique ability that the web has showed in permeating every aspect of our lives, becoming the perfect field for advertising campaigns. We can define in *Ads Providers* the second type actor of modern web.

In relatively recent years a new trend raised in the web advertising world based on the consideration that for adverts to be effective, reaching an high number of end users is not nearly as much effective as reaching those people who are more likely to be interested in the sponsored product. Thousands of services are born on the Internet that generate profits by collecting a wide variety of data and exploiting them to create complete profiles of users in order to target them with customized advertising contents, this phenomenon is known as Online Behavioural Advertising [2] and these services are known as *Web Trackers*, the third type of actor of the Internet ecosystem.

The last actor moving within this mutable world, are services able to exploit user profile data collected by trackers to deliver the most effective ads to each user. This services are know as *Ad Exchange* (AdX) and they implement a mechanism called Real Time Bidding Advertising involving all the actors identified so far [1]. We can illustrate this mechanism in a simplified way with an example: as soon as a user attempts to navigate towards the website of a content publisher he is recognised exploiting one of many techniques described below. Once he user is identified, it is possible for the website to access its profile data which are forwarded towards an AdX service, the ad exchange triggers an auction among the ads providers to gain the possibility of publishing their ads on the page rendered to the user. Suppose that this user is movie enthusiastic, the ads provider will be conscious about this thanks to the user profile built by trackers thus movie sponsors will submit their bids and attempt to gain a spot on the rendered page, the whole process including identification, auction and rendering, lasts only 10 to 100 milliseconds.

1.2 Web Trackers and Web Tracking

When a user interacts with a web page it is observed both by “first parties" that are the actual visited website and “third parties" which usually consist of hidden services gathering informations for analytics, social integration widgets, advertising and more. Third parties can reconstruct users' browsing history exploiting several techniques aimed to uniquely identify each user. In this section we analyse how these services work and which threats can derive from this practice.

1.2.1 Tracking Techniques

As said all tracking techniques are based on mechanisms aimed to uniquely identify users. The most common way to perform this task is by using *Cookies*. Cookies are described in [3] as a mechanism to manage state over HTTP: being HTTP, which is the transmission protocol which the great part of web communication is based on, a stateless protocol and thus unable to distinguish requests coming from different users, cookies were born as a mechanism to enable associating unique identifiers to users in order to recognize their future interactions by requiring web clients to present those identifiers at each request. As previously said, when users interact with a website they get in contact with a certain number of "third parties", usually these services host resources as HTML, images, JavaScript and CSS which are crucial for page rendering, any time the user's web client performs requests to retrieve these resources they can set and read previously set cookies.

Among tracking techniques, those based on the storage of some kind of string identifying users are known as Storage Based Techniques. Cookies go under this category and tracking techniques based on them have been developed in several different flavours. The major problem with cookie based techniques is that cookies are not persistent: some kind of cookies, known as session cookies, are deleted as soon as the browser is closed but even the so called persistent cookies have an expiry date, moreover it is quite simple for end users to manage and delete cookies on their web browsers.

During years trackers have found several ways to bypass this problem. Tracking methodologies have been developed that persist cookies by exploiting different storage locations, as Adobe's Flash Player Local Shared Objects (in this case we talk about Flash Cookies [4]) or web cache. By exploiting multiple storage locations and tracking techniques at the same time, cookies persistence capabilities have been dramatically increased up to the point of creating objects able to regenerate themselves that are extremely difficult to delete as demonstrated by projects like Samy Kamkar's "Evercookie" [5].

In addition to the cited ones, new methods of tracking are born that do not rely on any kind of cookie. It has been proven that it is possible to uniquely identify web clients by exploiting some browser features and plugins to create a fingerprint of the particular system which is running the client [6]. This fingerprint is not something users can easy delete as it is strictly related to the specific browser, plugins, hardware configuration and operating system used. Several techniques can be classified under this category which are based on a variety of mechanisms ranging from browser benchmarking [7], to the analysis of the way in which different web browsers render certain graphic elements based on <*canvas*> HTML tag [8].

1.2.2 Privacy Threats

Exploiting the described techniques, web trackers are able to associate a user to its browsing history which is tightly linked to a whole set of personal informations. The browser history can reveal our location, employment status, sexual and political orientations, health condition, shopping habits and preferences. The amount of collectable informations is not limited to the list of visited pages. Not only third parties are usually aware of the first-party page url from which a certain user has been redirected to them thanks to the HTTP referrer header, but they are also able to know the title of the original first-party page if it embeds scripts from the third party, in some case first parties voluntarily share even more informations.

Web tracking is not only a privacy threat: data collected with fingerprinting techniques can also be used to infer detailed informations about the tracked device hardware and software configuration. In case of single end users this can be considered a minor problem, but things are different for structured companies. Exploiting device fingerprints attackers could detect the usage of obsolete hardware or software versions and thus design specific attacks to target known vulnerabilities.

1.3 Protecting from Web Tracker

As described above, threats for users' privacy but also for cyber security systems coming from web tracking are real and constantly evolving. On the other hand there is no explicit policy describing what a website is allowed and what it is not allowed to track, as a consequence the web tracking phenomenon has often been growing within a regulatory vacuum [9].

Governments and developers are getting more and more conscious about the problem and working to find effective solutions. The recently approved General Data Protection Regulation - GDPR [10], clarifies what makes for valid user consent: consent must be freely given, users must be provided with specific and unambiguous informations about what that consent is about and approval must be indicated through an express affirmative act. Another effort in this direction is the DNT - Do Not track header field, developed as an efficient way to express users preferences regarding tracking in HTTP requests [11].

Although these attempts are necessary steps towards a solution, they are still far from being able to solve the problem. It is still difficult to perform an effective control over the correct and integral application of piece of legislation as the GDPR, while studies have showed that systems like DNT headers are often ignored by websites [12]. As a consequence many savvy users begun relying on client side tools to tackle the tracking problem. These tools most commonly come in the form of browser extensions, their working principle is simple despite the underlying implementation may be not. The idea is that of defining a set of rules to distinguish tracking and not tracking HTTP requests in order to block the latter while allowing the first.

Several studies provided evidences that these tools significantly outperformed other tracking protection methods such as the DNT header or the "Disable third party cookies" option present in several browsers [13], [14], [15]. Some of these tools also claim the ability of improving user quality of experience by reducing the amount of resources retrieved at page load and thus providing a speed boost to navigation [16].

1.4 Motivations

Despite ad-blocker and tracker-blocker tools resulted to be effective in countering the tracking problem, their adoption and use is not completely free from side effects.

As said, services acting as trackers are usually hosts of resources which, in some cases, are crucial for the first party page in order to be properly rendered and correctly work, as a consequence blocking all requests towards this kind of services can result in an unreadable or not properly working first party page.

The aim of this thesis is to examine a possible approach for the automatic and context independent detection of core contents of web pages, that is the part of the web page we do not want to be affected by tracker blocking tools in order to identify broken pages when using these tools. This document is organized as follows: chapter 2 goes through some related work in this field and discusses limitations of known approaches and differences with respect to the proposed solution, chapter 3 is about the design process of the implemented system and describes in details how it works and the reasons behind design choices, chapter 4 explains the test system design, the dataset used and how test data were collected, chapter 5 presents the results achieved by the implemented solution, finally chapter 6 evaluates critically the obtained results and discusses possible improvements and further developments.

Chapter 2

Related Work

The task of identifying core contents in web pages has been the research topic of several studies ever since the born of Internet. In fact solving this base problem is crucial to target several higher level problems. We might want to extract core page informations in order to provide input for the indexing of a search engine or in order to identify duplicated contents among web pages. Another common task is that of creating automatic tools for the adaptation of web pages rendering on small screens without the need of implementing display specific versions of the same page. Yet another application could be the evaluation of performances for tracker blocking tools, in order to understand if their adoption prevents the normal or anyway non functional rendering of the page. In general the tackled problem is that of eliminating the noise given by ads and non meaningful contents of pages, the approaches and techniques developed have undergone a long evolution process which we can consider almost parallel to that of the Internet itself and especially of web design.

2.1 Wrapper Based Approaches

During the first years of research the attempts to solve the problem used to rely on the implementation of site specific and partially hand-crafted tools called *wrappers* able of identifing data of interest and converting them into a more easily processable format. Several approaches have been developed to address the issue of generating accurate wrappers with small effort and in [17] a taxonomy of these methods is proposed despite specific implementations often take advantage of multiple techniques to accomplish their task.

Languages for Wrapper Development : one of the first attempts to address wrapper generation was the development of specific languages to assist the user in this task. An example of these intermediate mapping languages is WebOQL[1] [18], which is based on a middleware architecture for mapping source data in a common and more flexible data model. WebOQl exploits ordered arc-labelled trees called *hypertrees* to represent data, hypertrees can have internal or external arcs, the first used to represent structured objects the latter to describe references (usually hyperlinks) among objects. In this way it is possible to create an alternative representation of a web page in the form of a graph tree. Along with this data representation WebOQl defines a query language designed for performing complex restructuring operations over hypertrees and for hypertext navigation thus enabling restructuring page informations according to processing needs.

HTML-aware Tools : in this group we have those tools relying on the inherent HTML structure of pages in order to accomplish data extraction. These algorithms usually generate a parsing tree mirroring the HTML hierarchy of the page and then generate rules for the automatic or semi-automatic extraction of data from this hierarchy. This is the approach adopted in the W4F, the World Wide Web Wrapper Factory project [19]. W4F consists in a toolkit of different instruments able to generate wrappers defined through a declarative specification language. These wrappers have the capability of retrieving web contents, performing a clean-up of the HTML, parsing it into an abstract tree on which extraction rules are applied and finally mapping the extracted informations within a data structure suitable for the application purposes. The HTML extraction rules consist of declarative rules expressing a simple navigation along the tree and selection of pieces of informations.

NLP-based Tools : under this category we find those tools exploiting basic natural language processing techniques such as filtering, part-of-speech tagging and lexical semantic tagging in order to derive the relationships existing among phrases in a document and thus easing the derivation of extraction rules. One of the most known implementations of this paradigm is WHISK [20]. This specific tool exploits supervised learning to induce data extraction rules given a set of hand-tagged examples. Through the supervised learning process it is possible to learn guidelines which can be implemented into regular expression based rules able to recognize the context and the delimiters that make a certain phrase relevant for a certain domain.

Wrapper Induction Tools : are somehow similar to NLP-based solutions as they are capable of generating delimiters-based extraction rules given a set of training examples. The main difference is that delimiters are not identified according to linguistic or semantic constraints but rather according to structure

[1]Web Object Query Language

and formatting features of web pages able to implicitly suggest the relevance of the pieces of information they wrap. Tools belonging to this category are in this sense more oriented to HTML pages content analysis with respect to NLP-based solutions. A representative example of this approach is STALKER [21].

Modelling-based Tools : this kind of tools take as input the high level structure of any object of interest, then analyse the content of the page looking for structures conforming to the targeted ones exploiting algorithms similar to those used by Induction-based tools to locate content delimiters. The targeted objects are user defined combinations of primitive data structures such as tuples, lists, tables that allow a user to abstract from technical details of the targeted patterns like HTML tags or formatting operators which are more related to the low level implementation of the extraction problem. A well known application exploiting this paradigm is DEByE[22].

Ontology-based Tools : the last presented family of tools relies directly on data meaning rather than on their structure to perform data extraction. This approach exploits domain knowledge bases to support capabilities of semantic search, moving the analysis to an higher abstraction level with respect to plain keywords search. Given a specific domain it is possible to build an *ontology*, which is the set of concepts and relationships used to represent and describe that domain. An ontology classifies the terms that can be used in a particular context and characterizes possible relationships among them and eventual constraints in their use. Ontologies are used in a wide variety of fields, we bring an example related to the health care domain to clarify the concept. Supposing we have ontologies describing medical domains such as symptoms, diseases and treatments and pharmaceutical ontologies representiong informations about drugs, dosages and allergies, it is possible to combine these ontologies with patient data enabling a series of intelligent applicationb such as decision support tools for the selection of proper treatments [23]. In the context of information retrieval ontologies can be used combined with both keywords and semantic search techniques as presented in [24] to focus only on the page content that is actually related to the specific site domain as inferred by its ontology.

2.2 Methods Based on Templates

The main shortcoming with the automatically generated wrappers is given by their strict relationship with the addressed site, they lack in flexibility and are difficult to maintain as changes to page layout can require major modifications to the wrapper structure. This was not such a major problem during the first years of research in this field as pages were typically straight simple in layout and not evolving as fast as they do today but soon it became clear that more general purpose solutions were needed.

As Internet continued to grow and more and more websites appeared, a new approach was identified for core content detection relying on a new trend of web page design. Many web developers started exploiting *templates* to speed up the design of their websites. A template consists in an HTML page where both formatting and visual components are already implemented and ready to be filled with arbitrary contents. Pages implementing the same template share a common look and feel and are under the control of a single authority (as could be the website owner). Templates are useful in order to make the design process faster and allow component reuse among similar pages. They are also good from the user point of view as they make it easier to understand the browsing context thanks to a uniform and intuitive design.

Relatively soon, problems related to massive template adoption (it has been measured that a consistent part of all data present on the web is built exploiting templates), became evident. First of all they affect performances of crawlers: usually these piece of software exploit frequency and distribution of keywords and hyperlinks in order to rank a page, templates in most cases contain a large number of repeated hyperlinks and terms thus their large adoption might lead to inaccurate results of crawlers and indexers. On the other hand in most cases templates do not contain core pieces of the web page content, they instead implement page navigation and advertising sections thus they are noise in the context of core content extraction. From these considerations we can understand how templates detection can be considered a key point in identifying meaningful pieces of information within a page: once we get to know a template and we are able to assert with a certain margin of accuracy that a page is implementing that template we can easily find a way to extract core sections as we already know where to look for them. As a consequence we can see template extraction as something that is strictly related to content extraction, several approaches have been proposed to face this problem. We present now an example of template extraction technique.

2.2.1 Template extraction through hyperlink analysis

We will analyze the algorithm for template extraction proposed in [25]. In this work the approach is based on the analysis of the hyperlinks present on a web page. In simple words the implemented solution iteratively explores the set of hyperlinks contained in the tested page, let's call it P; then it looks for a certain number of pages which are mutually pairwise linked with P. The aim of this operation is to identify pages pointed by the website main menu as there is an high probability that those pages implement the same template. Once a set of pages S with these characteristics is found, their DOM [2] is evaluated to derive the template structure.

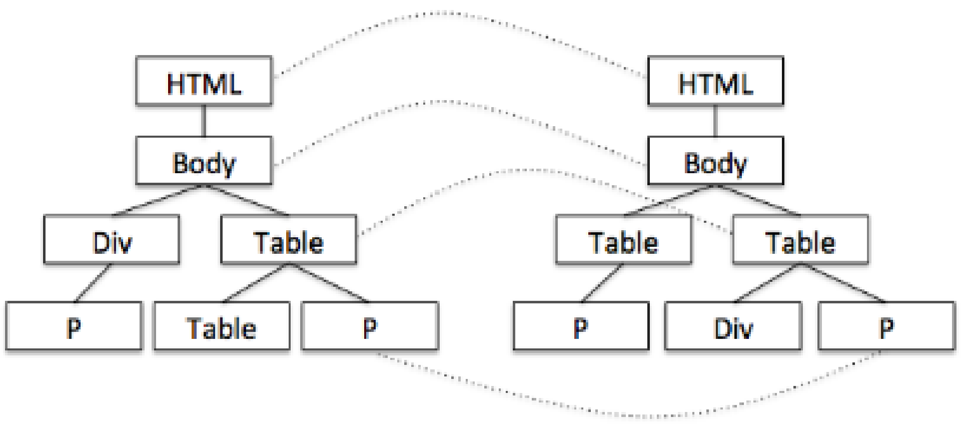


Figure 2.1: Example of ETDM comparison among two DOM trees.

In particular a technique called ETDM (Equal top-down mapping), illustrated in Figure 2.1, is used to evaluate corresponding equal nodes among the DOM trees of two pages. At the beginning the template model T is assumed to coincide with the DOM tree of the original page P, then the algorithm starts comparing T with one page extracted from the set S using ETDM. The comparison produces a set of DOM nodes which are identical among the evaluated pages, those nodes will constitute the new template model. The template model is refined by mean of subsequent iterations comparing the current model to new pages took from set S up to a point in which the model becomes stable. Once we get to know the DOM nodes belonging to the template we are able to assert that the core of the information is stored within the other ones.

2.3 Methods Based on Blocks

Several experiments have revealed how template based techniques can be effective, still there are many shortcomings related to them: first of all the main problem with templates is that they can be used for information extraction from a single specific kind of page, which does not necessarily mean that we are able to process all pages of a website with a certain template. In fact in modern Internet it is not unusual for the same website to implement multiple variations of the same page with different layouts and this could mean that multiple templates need to

[2]DOM stands for Document Object Model, it is an API allowing to represent an HTML page through a standard set of objects organized as a hierarchic tree. For a summary of current standards and specifications see https://www.w3.org/standards/techs/dom.

be identified to extract data from a single website, moreover today part of page contents and especially advertisements are generated at load time making template detection even harder. Another important downside with templates is that often the computational cost for the extraction of template models is quite high and it makes not possible to perform real time extraction of contents on newly visited pages.

Another family of data extraction techniques attempts to overcome the lack of flexibility of wrapper based and template based methods to achieve a general purpose solution to the extraction problem. These methods focus on splitting the page into blocks (which can be identified and delimited in several different ways) in a process that is called page *segmentation* and then apply some heuristic to decide if the obtained blocks contain or not useful informations. We can split this family of techniques in different subcategories.

2.3.1 DOM based approaches

In general this subgroup of techniques relies on hierarchical relationships among HTML tags to identify the basic blocks of a page. Each algorithm has a specific implementation of the segmentation mechanism but, in the vast majority of cases for DOM based approaches, it exploits particular HTML tags in order to identify blocks. In early days of Internet the most common way to organize the content of a web page was the <*table*> tag which, as a consequence, was considered one of the most effective block delimiters in many research works. With the development of HTML5 many pages started using <*div*> tags as well to organize content, as a consequence the segmentation task has been further complicated by the variety of possible design choices.

After this first step, DOM based methods exploit characteristic features of the extracted nodes in order to classify them as core or noisy contents. Several features can be used, probably the most commonly adopted one is *text density*. In [27] the authors start from an assumption:

> In a typical web page, (*i*) the noise is usually highly formatted and contain less text and brief sentences, whereas the core content is commonly simply formatted and contains more text but (*ii*) far less hyperlinks with respect to the noise.

Starting from this simple consideration the authors define their measure of text density for a node as the ratio between the number of characters in the node subtree and the number of tags in the same subtree. Content nodes will be the ones with highest ratio according to *i*. In order to take into account also *ii*, another measure is proposed called *Composite Text Density* which is computed considering also the number of characters in the node's subtree belonging to hyperlinks as well as the number of hyperlink tags in the subtree. In this way blocks with high text density but containing almost only hyperlinks as could be for example a list of links on the side of a page, are not wrongly classified as core contents.

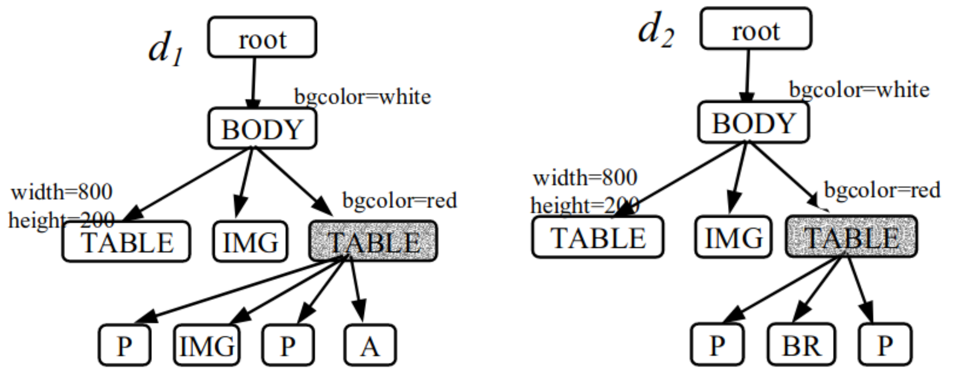


Figure 2.2: The examined DOM trees.

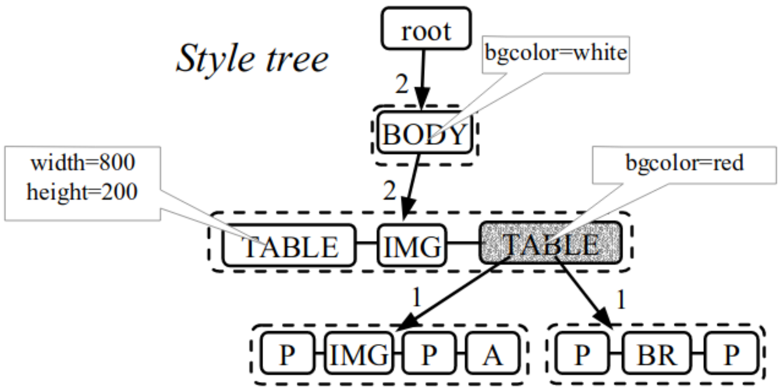


Figure 2.3: The resulting style tree.

Of course text density is not the only possible feature to be exploited and not always the most effective one for classification, in some cases also non core content of a page can contain high density text and a minimum number of hyperlinks. To make an example we could think to an advertisement banner containing the whole description of a rented house and a single link towards the related post on the house rental company website.

In [28] authors start from a different assumption with respect to [27] getting to a completely different approach to tackle the classification task. In particular they assumed that noisy (that is non content) nodes in a web page usually share not only the same content among all the pages of a website but also the same style and presentation. To exploit this particular intuition they developed a new way to look at a set of web pages by mean of *Style Trees* which are structures able to summarize the common presentation styles in a set of pages. In Figure 2.2 we can notice that in both pages at the second level of the DOM hierarchy there is the same sequence of nodes with the same style properties. In Figure 2.3, which shows the style tree representation deriving from the two pages, that specific sequence is represented as a single *style node*, the number 2 is the information about how many times that sequence appeared within the examined set of pages. Divergent sequences or sequences whose nodes have divergent style properties are represented by different style nodes as happens for the children node of the highlighted table nodes in Figure 2.2. In this sense style nodes represent different layouts and presentations of the page and convey also the information about how many times that particular combination of structure and style has been encountered. At this point the importance of a node is computed by evaluating the style tree along with the actual node content. The more regular and non mutable the content and style are the higher the probability for that node of containing non relevant informations.

2.3.2 Semantics based approaches

Semantic based approaches perform page segmentation according to keywords and text features of the node's content rather than according to its HTML structure and position in the DOM hierarchy. As reviewed in [29] there are several works exploiting resources and techniques typical of the Semantic Web [3] world (such as languages, ontologies and knowledge-bases) to improve information extraction systems performances and vice versa there are also researches that exploit information extraction techniques in order to populate the Semantic Web.

An interesting example of this family of approaches is the one related to the *Artequakt Project* [31]. In this work the authors propose a complete architecture aimed to the collection from the web of data about artists, the creation, storage and management of these informations within a *knowledge base* [4] and the automatic generation of biographies of artists in which the collected data are organized and presented in a narrative way. It is interesting to analyse their approach to the information extraction problem as it is not strictly focused on the mere content extraction rather on the identification of the entities related to their specific domain and their relationships.

When a user queries for a certain type of biography on a certain artist, the

[3]Semantic Web is an extension, proposed by W3C the World Wide Web Consortium, of the current paradigm of the web which is made of isolated documents connected by simple hyperlinks. The new paradigm, which is implemented through a full stack of new technologies, enables a web made of data connected by meaningful relationships, similarly to what happens in modern databases. This change in perspective would open great possibilities in terms of automation and intelligent applications on the web, see W3C definition at Semantic Web.

[4]A knowledge base KB is a particular kind of storage system that differs from common databases. The informations stored in a knowledge bases represent the concepts and relationships related to a certain domain in the form of classes, subclasses and instances of these classes all linked by mutual relationships.

system performs a web query on the web and explores the resulting websites. If a trusted web resource is known related to the domain, the system looks for it as first step and then uses data extracted from this resource to filter the other resulting web resources according to similarity measures. To retrieve informations from a website the Artequak system extracts all paragraphs from it, the text extraction can be performed using one of the known techniques discussed as for example the text density based approach. Then once the core content extraction step has been performed an additional processing is conducted evaluating contents at semantic level. First a parser is used to group grammatically related phrases thus a syntactical analysis is performed. At this stage the semantic analysis starts: the semantic components of the phrase are identified through text processing tools as GATE [5], words are classified as subjects, verbs or objects and named entities are recognized as persons, places and in general instances characterized by a proper name of some conceptual class . At this stage the system exploits the knowledge base to discover and extract relationships among concepts, the domain knowledge is crucial in this stage as it suggests which entities could be linked. What is extracted is not simple text but a structured knowledge related to the queried artist which allows us to store in a machine friendly way that *Rembrant* which is an instance of the class person is linked by a *date of birth* relationship (which is a relationship owned by all instances of the class person) to the entity *1606 July 15th* which, on the other hand, is an instance of the class date. This kind of content is much more useful and easy to process with respect to the mere text.

2.3.3 Vision based approaches

The last family of techniques we are going to analyse exploits the visual characteristics of the rendered page in order to extract its content, these characteristics include but are not limited to font type size and color, background and visual tags. Combined with the information coming from the DOM hierarchy these data can ease the process of segmentation as often the core contents of web pages are similar in the way they are styled and presented. An immediate example could be font size analysis, it is quite common for websites to use font sizes between 14 and 16 px for main content as it is easily readable without taking too much space on the page. Evaluating the presence of similar commonly adopted choices provides hints about how to classify blocks. On the other hand the visual approach is very effective for the segmentation task as usually different or unrelated information blocks in a page are styled in different ways. Another important aspect is the tendency of web interfaces to always present particular functionalities in certain, well known to users, positions. In order to ease the navigation process many website adopt a

[5]See https://gate.ac.uk/

common spatial distribution of functionalities and, as it has been showed in [32], users expect this distribution.

In [33] this kind of approach is applied specifically to the problem of page segmentation, the developed algorithm is named VIPS - VIsion-based Page Segmentation. It exploits at the same time visual and DOM characteristics to subdivide pages into blocks. The implemented solution performs a top down analysis of the page in a row of subsequent rounds. At each round the page is segmented into blocks corresponding to DOM tree nodes, then for each of these blocks a score is computed called *DoC* which stands for Degree of Coherence, this score is once again based on DOM characteristics but above all on visual cues and represents a measure of how much the content of the block is coherent with itself. This is an important information as the more internally coherent a block is, the more likely it is for it to be a semantically autonomous section of the page. According to the desired granularity of the analysis a threshold is set to state the minimum DoC score for a block to be considered independent, at each iterations all blocks not meeting the requirement are added to a pool and are further analysed and split in subsequent rounds.

To decide the SoC score of a node and in general to decide if it is necessary to divide a block several aspects are considered, the most important cues exploited can be classified into three categories:

Color cues - It is preferred to divide nodes whose children present a background color differing from the one of their father.

Text cues - It is preferred to avoid dividing a node in case all of its children nodes are *text nodes*, that is they contain only free text without any html tag. These kind of nodes are assigned the maximum DoC score (10) in case the font size is the same for all text nodes.

Size cues - A relative size measure is defined that evaluates how much a node is big compared to the whole page or sub-page which it is part of. A threshold is defined that varies for different html tags as of course a <*table*> node is typically bigger than a <*a*> node. If the relative size of a node is under the threshold for that type of tag, it is not divided. On the other hand if the sum of the sizes of the children nodes of a block is greater than the size of the block itself, it has to be divided.

2.4 Limitations of Known Approaches

As we can state from the number and the variety of the studies performed in this field, a lot of work has been done to find an effective solution to the crucial task of distinguishing noise from core content. As seen, the first attempts to solve the problem were based on custom wrappers which were designed and implemented with the help of semi-automatic tools, their main disadvantage was their low maintainability and flexibility. It was relatively complex to design such wrappers and the wrapper generator itself were not able to stand against the rapid evolution of modern web in which the technologies and standards of web design change at an impressive peace. On the other hand methods based on templates were found to be more general purpose as the practice of providing a general and reusable template has become a milestone in modern websites design. Still this kind of approaches needed to build some knowledge for each specific website before being able to parse it, this meaning that no real time application were possible for new websites. Moreover modern websites often exploit multiple templates and it is also possible to have multiple layouts for the same page.

Methods based on page segmentation into blocks have been proved to be effective and more generally applicable even in case of pages visited for the first time. The approaches based on DOM tree analysis have achieved important results but nowadays with the development of more and more complex frameworks able to generate HTML contents at runtime the complexity of DOM structures is not limited to what a human developer can create and maintain and it becomes more and more difficult to locate core contents among hundreds of nodes. Semantic based approaches are very effective in locating contents but rely on knowledge bases that are difficult to create and maintain, moreover they require natural language processing techniques to evaluate page content before being able to classify it and this usually is a time consuming task.

In this work we propose a solution that could be classified among the visual approaches of the block based methods. The intent is not to extract the core content but to visually locate the sections of the page that contain that informations in order to automatically notice eventual problems in the page rendering. The developed approach is general purpose, DOM independent and and achieved good results in terms of accuracy and computation time.

Chapter 3

Proposed Solution

In this chapter we discuss in depth the different aspects of the developed solution. First a comprehensive overview of the solution design is presented in section 3.1. Then in section 3.2 the system exploited to interact and collect data from the web is analysed. After that the two algorithms on which the solution is based are introduced and discussed in terms of funding principles and actual implementation respectively in section 3.3 and section 3.4. In order to ease the reading of this chapter, in Appendix A we present two tables summarizing for both algorithms the involved parameters, with their names and a brief description of their role.

3.1 Solution Overall Design

The whole system design started from a simple question: how is it possible to understand that a page has not been properly rendered? To answer this question we had to identify which was the main characteristics distinguishing a normal page from a broken one. The answer was quite simple, a page is very unlikely to be "broken" if its main content, its navigation bars and in general all the elements of that page that are related to its meaning or to its functionalities are rendered correctly; as a consequence we decided to focus on the characteristics of this kind of elements.

The most important aspect with those components that provide functionalities to the website is that they tend to maintain always the same in shape and style: if the page is visited multiple times within a limited amount of time, it is possible that multiple layouts (or templates) will be rendered thus obtaining different html structure, nevertheless the main functional elements and the core content will always be rendered in the same way although they could change their position within the page. On the other hand the advertising elements which usually constitute the great majority of the non functional elements within the page are, for their inherent nature, always changing both in shape and size (depending on the rendered layout

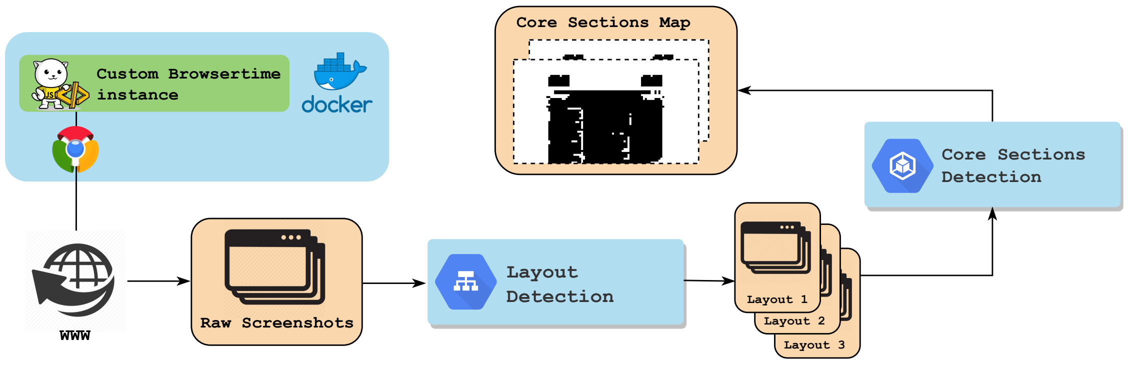


Figure 3.1: Schema representing the whole system pipeline

) and in their content as well.

Given these assumptions, our problem can be reduced to locate on the page the elements which are not changing in form, shape and above all content. Starting from the previous works we oriented towards a method based on blocks, this is a natural consequence if we think that our task is actually that of locating blocks of contents which are always equal to themselves across multiple visits. As seen in section 2.3, block based techniques can be divided into three families: DOM based, semantic based and visual based approaches. Of course it was not possible to exploit a semantic approach in our case, being the objective not only to locate the content of the page independently from the semantic contest, but also to identify all its functional and thus non semantically meaningful elements (we can think to an arrow shaped button for example). DOM based solutions have been evaluated to be unfeasible as well, the complexity of DOM trees and the variety of possible solutions in modern websites, makes it hard to identify a general reproducible solution. Thus we decided to adopt a visual approach in which, given two captures from the same page stating if a certain section of the page is equal in both images is as easy as subtracting the two sub-images representing that specific section and evaluating in the resulting image how many pixels were subtracted to 0.

In Figure 3.1 we can see a schema describing all steps of the designed pipeline. The system consists of three main components which live independently from one another and perform their specific tasks in an autonomous way.

The first component is the one in charge of collecting the input data from the web. This component is wrapped in a docker instance and consists of a modified instance of Browsertime. This component is able to instantiate a web browser in order to visit and navigate websites, collecting several types of metrics as we will further discuss in section 3.2. In our implementation Browsertime drives an instance of the popular web browser Google Chrome [1] visiting a set of predefined

[1]https://www.google.com/chrome/

web pages and collecting, along with other metrics, a whole set of screenshots of the rendered page.

Once we get to have a set of screenshot of a certain page we need a further processing before being able of locating the core content sections. As already said in the modern web it is not unusual to visit a web pages ten times getting rendered six different layouts of the same identical page, this is mostly due to different sizes and positions of the advertising contents. The Layout Detection Algorithm presented in section 3.3 is able to identify the different layouts and cluster the set of screenshots into groups sharing the same page structure.

At this point we simply select the most representative cluster, that is simply the one having the greatest number of captures. The reason for selecting the biggest cluster will be more clear after the introduction in section 3.4 of the section detection algorithm, in simple words the precision of the static sections grows when the number of analysed captures increases. The screenshots associated to the selected cluster are processed by the Section Detection Algorithm detailed in section 3.4 in order to locate the static and dynamic sections. The output of this process is a image having the same dimensions with respect to the original screenshots. This image shows the dynamic sections coloured in black and in white the static ones, because of this characteristics it can actually be used as a mask to extract the core sections from the original captures by performing a pixel to pixel product with the original captures.

3.2 Data Collection

In this section we will further analyse the technologies and tools exploited during the data collection phase. First we are going to introduce Docker and its capabilities, then we are focusing on Browsertime, the reasons behind its adoption and the modifications we had to perform to adapt it to our needs.

3.2.1 Docker

Docker is an open source platform [2] created with the aim of easing the development, delivery and deployment of pieces of software. Modern application often rely on a whole set of hardware and software dependencies and require several different configurations for each phase of their life cycle. Keeping everything on a single machine can result in incompatibility problems among different components and make it hard to migrate our application to a different environment for testing or production. There are two approaches to solve this kind of problem: host virtualization and software virtualization. The first approach exploits a low level

[2]See https://www.docker.com/why-docker

component called hypervisor able to handle physical resources of the host machine and to host virtual systems by duplicating the whole OS and virtualizing the hardware resources. The second approach exploits directly the host operating system by creating a new process which is assigned its own namespace, in this way it is isolated from the rest of the system and can access operating system resources associated to this namespace whose implementation can differ from the one of host machine. Docker exploits this principles in order to enable creating autonomous environments called *containers*. The communication among the host machine and the container is performed through a command line interface that interacts with a rest API on which a container long-running process called daemon is listening.

The main advantage in using docker is that you can create a completely new environment with specific versions of each component without affecting in any way the host machine configuration. This also means that it is possible to move or duplicate that environment from one container to another without any compatibility problem.

Docker containers are distributed in the form of images which can be seen as read-only templates providing all the instructions needed in order to create a container starting from the operating system setup and configuration, up to the installation of any application of interest and of its dependencies. Images can be derived from other images adding slight modification to the system configuration or installing new dependencies; this is possible because docker exploits a layered file system called Union FS which is made of multiple read-only layers, thus extending an existing image is as simple as adding an new layer to the stack. Images are described through text files called *Dockerfiles*.

The docker image used for our data collection environment is derived from the Browsertime Docker image [3] which has been slightly modified, we can see its dockerfile in listing 3.1. First a new version of chrome has been installed on the base Browsertime image, the resulting image has become the basis for further modifications (line 1). Then some source files of Browsertime have been substituted to achieve custom behaviours (lines 6-9) as we will discuss in depth in the next section. Finally a folder containing custom chrome profiles has been created in the container file system (line 12-13).

```
1 FROM browsertimebase_chromev71:latest
2
3 WORKDIR /browsertime
4
5 # Overriding customized files of Browsertime src
```

[3] Dockerfile at https://github.com/sitespeedio/browsertime/blob/master/Dockerfile

```
COPY mods/lib/support/getViewPort.js
    /usr/src/app/lib/support/getViewPort.js
COPY mods/lib/core/seleniumRunner.js
    /usr/src/app/lib/core/seleniumRunner.js
COPY mods/lib/screenshot/defaults.js
    /usr/src/app/lib/screenshot/defaults.js
COPY mods/lib/screenshot/index.js
    /usr/src/app/lib/screenshot/index.js

# Adding chrome configuration files with custom profiles
RUN mkdir -p /tmp/rep
COPY /config/chrome-new/* /profiles
```

Listing 3.1: Dockerfile for custom Browsertime container.

3.2.2 Browsertime

Browsertime is part of a family of open source tools that goes under the name of *sitespeed.io* [4], these tools are designed for monitoring and testing performances of web pages by collecting several types of metrics. It enable to automate the process of visiting and navigate web pages by exploiting the Selenium WebDriver. Selenium [5] is a tool for browser automation, that enables performing through code every kind of interaction a user would normally be able to accomplish; its WebDriver is designed to provide a common interface for driving browsers, it is supported by Chrome and Firefox implementations allowing to drive these browser in a natively way.

Browsertime has been selected among other similar tools because of several interesting features:

Open source : being an open source project it is possible to access its source code and modify it with ease.

Several metrics available : Browsertime is highly configurable and allows to collect a wide range of metrics going from HAR [6] files to timing and visual metrics for measuring page rending performances.

[4]See https://www.sitespeed.io/

[5]See https://www.seleniumhq.org/

[6]HAR stands for HTTP Archive, it is an evolving standard of an archival recording, in json format, for HTTP transactions. Within an HAR file you can expect to find several metrics such as the time to retrieve each object of the page, time required to fetch DNS informations for site, time spent to establish a connection with the server etc. The current standard is available at https://dvcs.w3.org/hg/webperf/raw-file/tip/specs/HAR/Overview.html

Screenshots and video : Browsertime uses internally FFmpeg [7], a famous tool for recording, converting and streaming audio and video, to enable the collection of screenshots and videos of the visited pages.

Native docker support : as we have already mentioned Browsertime is available as a ready to use docker container, easy to run, modify and migrate.

Support to Web Page Replay Go : WprGo is a tool integrated in Browsertime from version 3.0 enabling to record a certain visit to a web page and reproduce that particular visit when desired. It acts as a sort of proxy that caches all the webpage resources and records all the http requests becoming able to repeat the exact original HTTP transaction. This feature is very useful for testing as it allows to capture the particular page served by a website in a certain moment and visit it in the same exact way multiple times.

Some slight modifications were required to the out-of-the-box version of Browsertime to be adapted to our specific needs. First of all it was necessary to make the visual rendering of our visits independent from the host machine running it, thus the support scripts *getViewport.js* has been modified to return always the same default viewport, the value returned by this script is provided as parameter when instantiating the browser and affects the width and height of the browser window. Also the scripts for the screenshot capture at line 8 and 9 of listing 3.1 have been modified to avoid the default resizing operations performed over the captured images. The result of all these mods is a that Browsertime independently on the hosting system returns screenshots of size 1920x1080px, this has been done in order to provide a uniform set of inputs to the subsequent components of the pipe.

```
// [...] code checking for completion of page load
/*
Code simulating the implicit acceptance of cookie policies
Simply scrolls the page down and up again
*/
delay(1000);
this.driver.executeScript('window.scrollBy(0, 1000)');
delay(3000);
this.driver.executeScript('window.scrollTo(0,0)');
delay(1000);
// [...] code in charge of taking the page captures
```

Listing 3.2: Modification to seleniumRunner.js.

Another problem faced during the data collection step was that of banners requiring to enable cookies. The purpose of our work is to identify static (core)

[7]See https://www.ffmpeg.org/

sections and dynamic ones, the latter are usually made of advertising. In the vast majority of cases when visiting a website for the first time a banner pops up requesting to accept the cookie policy of the website, in order to render the page normally and taking a meaningful screenshot of it, it was necessary to find a way of providing consent to these requests. To accomplish this task a minor modification of script *seleniumRunner.js* was required. This script is in charge of instantiating the web browser, navigating to the target page and waiting until it is completely loaded while collecting metrics on it. We modified it by inserting few lines of code to be executed immediately after the page has been loaded and before capturing the screenshot. In several websites scrolling the page is considered to be an implicit acceptance of cookie policies thus the simple snippet showed in listing 3.2 made the trick.

The final output of the data collection step is a sequence of screenshots of standard dimensions 1920x1080, which show multiple rendering of the same page collected during different visits performed within a limited amount of time (in the order of tens of minutes). A sample of the output of this first step is visible in Figure 3.2.

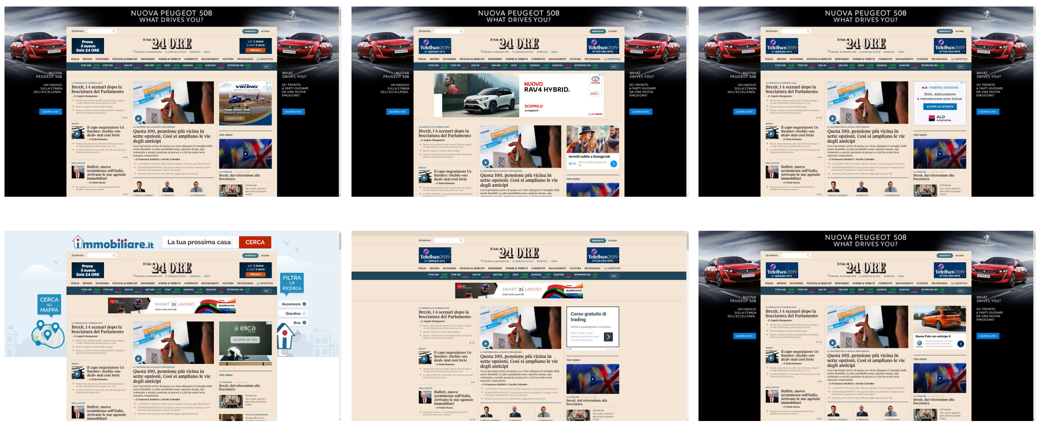


Figure 3.2: Sample screenshots from visits to ilsole24ore.com

3.3 Page Layout Detection Algorithm

As it is evident by observing captures in Figure 3.2, when we visit multiple times the same page we obtain different page layouts. Pages with differing layouts are characterized by a different number of visible rendered elements as well as by a different organization and shape of advertising contents. Despite of this the captures obtained are still representing the same page, the main content does not change among visits which are close enough in time and neither do the navigation elements, the page logo and in general all the static elements of the page. Before being able of identifying and locating the static content we need a set of screenshots having the same layout in which only the dynamic and advertising content changes while the static contents remain in the same fixed position of the page. We need to identify the possible layouts and group our captures according to them.

The task tackled is quite a peculiar one which, as far as we know, has never been addressed in any previous research work. First we performed attempts to readapt known clustering algorithms such as DBSCAN [34] to our problem. We found that defining a good distance measure for two screenshots, stating their probability of belonging to the same layout is an highly complex task not only for the challenge of reducing an articulated problem to a single measure but also in terms of computational time. For all these reasons we decided to develop a completely new algorithm from scratches, which would have been targeted to our specific problem and able to exploit the knowledge related to its specific domain. The developed solution has achieved high levels of accuracy and good performances in terms of computational time.

The algorithm has been implemented in Python 3[8], this choice was an almost obliged one due to the great amount of libraries and utilities for image processing available. Before getting into the details of the developed solution, we introduce some terms that will be used in the algorithm description:

- **layoutCluster** : is the term that will be used to refer to a group of screenshot with the same layout. Within the algorithm it is represented by a python dictionary object with multiple properties as we will discuss later on. We will refer to them also as clusters or templates or layouts.
- **layoutsPool** : is a dictionary object containing all the layoutClusters created at a certain point of the algorithm execution.
- **capturesPool** : is a list keeping all the captures that have not been classified at a given instant during the algorithm execution, to each capture some meta data are associated which maintain eventual knowledge acquired about this capture during previous iterations.

3.3.1 Algorithm workflow

The developed algorithm adopts an iterative approach, at each iteration it evaluates all the unassigned screenshots and attempts to assign them to one of the layoutClusters discovered so far and maintained in the layoutsPool.

Before starting the first iteration a preliminary computation is performed on all captures. Each screenshot is subdivided into sub-images representing a section of the original image whose shape is defined by the algorithm parameters *blockWidth*

[8]See https://www.python.org/

and *blockHeight*. What we keep in the capturesPool is not the capture itself but the array of the sub-images, that we will call *blocks*, derived from the splitting operation on that capture. The dimensions of the block are important parameters in the algorithm economy as they allow to define the granularity of the analysis, in fact every time we will compare two captures we will actually compare the corresponding blocks derived from the two images.

At the very beginning of the execution workflow all captures are contained within the capturesPool while the layoutsPool is empty. After all captures have been converted into arrays of blocks, we pick the first capture in the capturesPool and create a layoutCluster object containing it as the only representative element, then we add this object to the layoutsPool.

```
{
  "tpl0": {
    "captures": [capture1, capture2, ...],
    "flagChanged": False,
    "previousIterationLength": 0,
  },
  "tpl1": { ... }
}
```

Listing 3.3: Example of layoutPool object.

In listing 3.3 we can see how the layoutsPool is a simple map associating the layout name to each layoutCluster object. A cluster object is itself a python map with three named properties. The **captures** property is a list of the captures assigned to the layoutCluster so far, captures are added to the end of this list as soon as they are assigned to the cluster. The **flagChanged** property is a boolean stating if this cluster has been modified (meaning that new captures have been added to it) during the last iteration of the algorithm. Finally the **previousIterationLength** property is a number keeping track of how many elements were stored in the captures list during the previous iteration thus, in case any element has been added to the cluster in the last iteration, it can be used to identify the first new element in the captures list. Clusters are given a name and added to the pool as soon as they are created, after their insertion they cannot be removed but only modified by adding new elements to the cluster.

Clusters can be created only in three different situations:

1. During the preparation step: as said, the first cluster is created immediately after all captures have been converted to array of blocks using the first element of the capturesPool which will become the first element in the cluster captures property. Its flagChanged property is set to True and the previousIterationLenght is set to 0.

2. When during the last iteration no cluster has been modified: in case no cluster

was assigned new captures during a whole algorithm iteration a new cluster is created by picking the first element of the current capturesPool, its properties are set in the same way we do for the first cluster.

3. In case a screenshots is evaluated to be unmatchable with all the currently known clusters in the layoutsPool (the mechanism to state this condition will be discussed later on). The new cluster is created using the unmatchable capture as first element and initialized in the same way we do for the first one.

After the preparation step the main loop of the algorithm can begin, its flow is very straight forward and is summarized in listing 3.4. As said, we iterate until there are no unassigned captures remaining, at the beginning of each iteration we update the control properties (flagChanged and previousIterationLength) of each layoutCluster object. Then we run the clustering routine which can modify the layoutsPool by assigning captures to one of the already known layoutClusters or even by creating new clusters. The return value of the routine is the new list of unassigned captures. We get to know if some screens have actually been assigned by simply comparing the length of the newCapturesPool with respect to the previous iteration capturesPool. In case some screens have actually been assigned the capturesPool is update, instead, if no progress has been done, we create a new cluster in the way described at point 3 of cluster creation methods and update both the layoutsPool and the capturesPool.

```
while( len(capturesPool) > 0 )  {
  # Updates flagChanged and previousIterationLength
  # for each layout by checking the length of their captures array.
  Update(layoutsPool)

  # Here the actual classification is performed
  newCapturesPool = PerformClustering(capturesPool, layoutsPool)

  If( len(newIngPool) < len(imgPool) ):
    # Some screen have been assigned
    capturesPool = newCapturesPool;
    continue;

  else:
    # No screen assigned, creating a new template
    capturesPool = CreateNewTemplate(capturesPool, layoutsPool)
}
```

Listing 3.4: Main loop of the layout detection algorithm.

The main core of the algorithm is the clustering routine which is presented as pseudocode in listing 3.5 and listing 3.6. It iterates over the pool of unassigned captures and, for each of them, we iterate over the pool of known layouts (line 6 in listing 3.5) attempting to associate them with one of the known templates in the *EvalCptAgainstLayout* routine (line 16 in listing 3.5). In order to assign a screenshot to a particular template we compare it with the representative elements of that template. We compare the tested capture only one time with each representative element of a layout being the result of this evaluation deterministic, the comparison is performed during the subsequent iteration with respect to the one in which the rep was added to the cluster. There are three possible outcomes when we compare a capture with a rep from a certain layout:

- **Compatible layouts**: we find the capture to be compatible with the examined layoutCluster. We assign the capture to the layoutCluster and skip to the next capture (line 31 in listing 3.5).
- **Non compatible layouts**: we find that in any case this capture can not be assigned to this layout. We record this information in the capture object metadata and skip to the next layoutCluster (line 34 in listing 3.5).
- **No result**: we are unable to state if the capture is compatible or not with the layout. We go on and examine the next rep of the same layoutCluster.

```
 1 newCptPool = [ ]
 2 while( len(cptPool) > 0 )  {
 3   testImg = cptPool.pop()
 4   imgMetaData = GetDataForImg( testImg )
 5
 6   for( layout in layoutPool )  {
 7     # Avoid processing incompatible templatess
 8     if isNotCompatible(imgMetaData, layout):
 9       continue
10
11     # Select only members added in previous iteration
12     newTemplateMembers = GetNewMembers( layout )
13     lytFound = False
14     noMatch = False
15
16     lytFound, noMatch = EvalCptAgainstLayout(testImg,layout)
17
18     if ( lytFound ):
19       AssignScreenToLayout( testImg, layout )
20       break
21     if ( noMatch ):
22       recordIncompatibility( imgMetaData, layout )
23   }
24
25   noMatchCount = CountIncompatibleLayouts( imgMetaData )
26   if ( noMatchCount == len( layoutPool.keys() ) ):
```

```
27     CreateNewTemplateForImage( testImg, layoutPool )
28   else:
29     # Attempt to assign this image during next iterations
30     newCptPool.add( testImg )
31 }
```

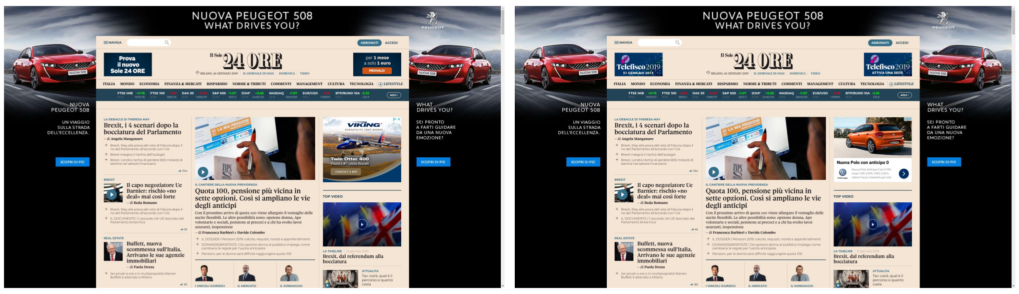
Listing 3.5: Clustering routine pseudocode

Among what is stored within the capture objects metadata, the information about the incompatibility of the screenshot with respect to a certain layout is very important for two reasons: first, if we know that a capture will never be matchable with a layout we avoid comparing it with the reps of that layout even if new elements were added to that cluster during the last iteration (line 8 in listing 3.5). In second instance after evaluating our capture against the reps of a layout we check if the number of layoutClusters which are incompatible with it is equal to the number of known clusters, in that case, according to point 2 of cluster creation methods, we create a new template from that capture (line 25-27 in listing 3.5). In case after the evaluation of our capture against all current reps of all current layouts neither a compatible layout has been found nor an incompatibility with all of them has been assessed the capture is added to the new pool of captures to be used during the next iteration (line 30 in listing 3.5).

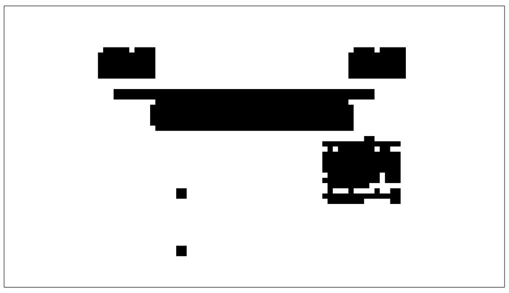
```
# Select only members added in previous iteration
newTemplateMembers = GetNewMembers( layout )
lytFound = False
noMatch = False
for( lytImg in newTemplateMembers ) {
  # Block to block subtraction
  changesMap = ComputeChangedBlocks(testImg, lytImg)

  lytFound, noMatch = Step0_HighLvlSimilarity(changesMap)
  if ( lytFound || noMatch ) break

  lytFound, noMatch = Step1_CoreLvlSimilarity(changesMap)
  if ( lytFound || noMatch ) break

  lytFound, noMatch = Step2_SearchPattern(changesMap)
  if ( lytFound || noMatch ) break
}
```

Listing 3.6: Routine evaluating a capture against a layout

Now that we have a comprehensive idea of how the execution flow of the algorithm works, we can examine in details how is implemented the evaluation process, that is where decisions about the captures are taken. The evaluation steps are summarized in listing 3.6. As said the idea is that of comparing a capture against each rep from a certain layout, looking for hints enabling us to assert whether the capture should or should not be added to the same cluster of the tested rep. The first step of this evaluation is the *Perform Comparison* routine that takes as input the tested captures which, as said, have been previously converted into an array of sub-images that we call blocks. The routine workflow is quite simple: first a pixel by pixel subtraction is performed among the corresponding blocks of the two images and the absolute value of the result is computed. Then we count how many pixels of the resulting blocks are equal to zero, if the number of pixels whose value is not equal to zero is greater than a certain threshold *NN0 - Number of Non 0*, which is expressed as a percentage and thus is independent from the block dimensions, we state that the block has changed, otherwise we assume that the sub-images represented by those blocks are identical. The result of the comparison process is an array of boolean values with as many elements as the blocks composing each of the compared captures; this array can be used as a map telling us which sections of the evaluated images differ and which not. This map, that we call *changesMap*, becomes more and more accurate as we reduce the dimensions of our blocks but the computation time required to produce it increases as well.



(a) Compared images



(b) Map of changed blocks.

Figure 3.3: Example of comparison among two images with the resulting map.

In Figure 3.3 we can see an example of comparison among two captures and the resulting changesMap which is represented as an image obtained colouring in white those blocks which were evaluated to be identical and in black the ones differing in the two images. From this point on we will refer to the identical blocks as *static* *blocks* and to the differing ones as *dynamic blocks*. After we obtain this sort of map of identical and differing blocks, it becomes the inpt for three routines that perform the actual evaluation of the capture by exploiting this map to derive hints suggesting whether the examined capture belongs to the same template of the rep to which it has been compared. We will now discuss in depth how these routines work.

3.3.2 High level and core level similarity

The first kind of hints we look for are related to similarity measures performed on the compared captures. The basic idea behind the evaluation of these measures can be expressed by two complementary considerations:

> (*i*) Captures from different layouts are more likely to show an high percentage of differing blocks as, when the layout changes even the static sections of the page are translated and thus are evaluated as dynamic blocks.
> (*ii*) Captures representative of the same layout have high probability of showing low percentages of differing blocks as the static content of the page will always be counted among static blocks.

Assumptions ***i*** and ***ii*** are the base principle of functioning of the high level similarity routine (line 9 of listing 3.6). What we actually do is computing a measure called *Degree of High Level Similarity* (DHLS) between the evaluated captures. This measure, calculated exploiting the changesMap, is simply the percentage of non differing blocks over the total. Once computed the DHLS value for the captures we check it against some thresholds and we set the lytFound and noMatch flags to be returned according to the outcome of this check:

- **DHLS < minHighLvlSimilarityTh**: the evaluated captures are so similar that, according to ***i***, they must be representative elements of the same layout.
- **DHLS > maxHighLvlSimilarityTh**: the evaluated captures are so much different that, according to ***ii***, they cannot be representative elements of the same layout.
- **minHighLvlSimilarityTh < DHLS < maxHighLvlSimilarityTh**: we are not able to state anything about this capture with respect to this template.

An example of images classified according to this principle is showed Figure 3.4 along with their changesMap. Unfortunately in many cases the high level similarity routine is unable to take a decision about a capture. The reason is that given the growing space advertising and dynamic contents have in modern web pages,

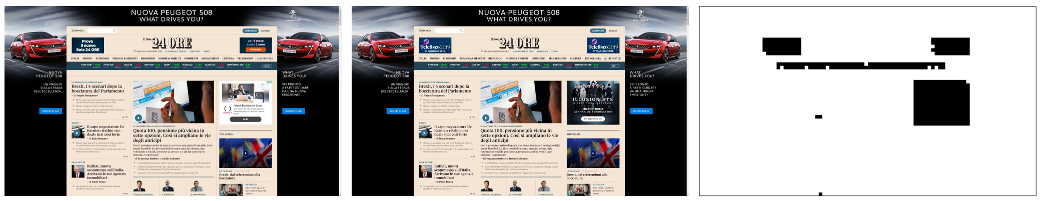


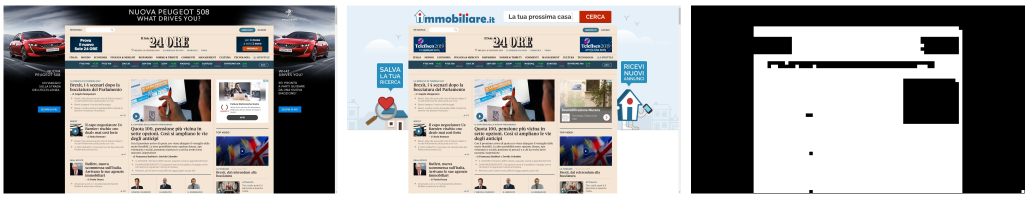
Figure 3.4: Captures classified using High Level Similarity and the related changesMap

often we can have pages rendering the same exact layout but differing in a great number of blocks. In Figure 3.5a we can see two captures exemplifying this concept. Both pictures implement the same page layout with a big advertising banner as background and a smaller banners on both sides of the title. The background banner takes a considerable amount of space, if we create a changesMap of these two picture we will find that their DHLS value is in the range [0.4, 0.6], in other words it is difficult to state anything about these captures as they are neither enough similar nor enough different to provide useful hints; in this case the DHLS fails in providing useful informations. This kind of situation is quite common in modern websites but we can exploit the knowledge deriving from our domain of interest in order to reduce the noise produced by these types of layouts. In particular, we can increase the decision capabilities provided by ***i*** and ***ii*** considering that:

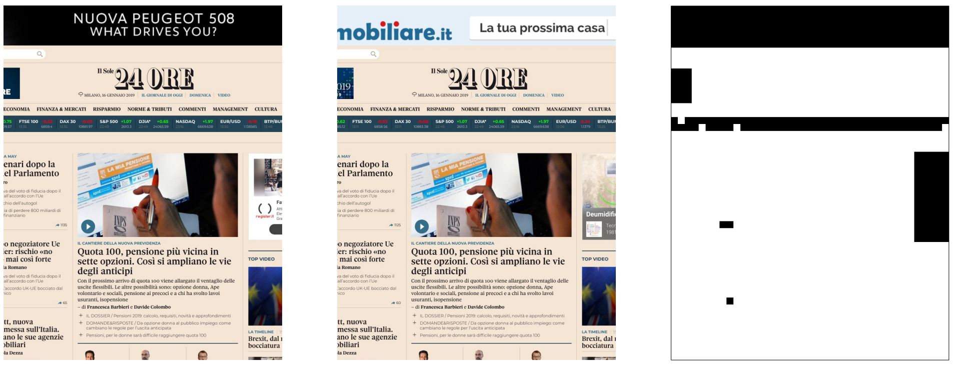
> (*iii*) The central part of a web page is typically the one containing the vast majority of static sections and thus is the place in which it is easier to spot differences due to page layout.

The simple consideration in ***iii*** is the funding principle of our second classification step: the core level similarity routine (line 12 of listing 3.6). This routine computes a measure called *Degree of Core Level Similarity* (DCLS) that is in many ways similar to the DHLS measure as it is computed again as the percentage of static blocks of a changesMap. The difference is that rather than being computed on the whole changesMap, it is calculated considering a limited section of it which is delimited by a window positioned in such a way to select a portion of map corresponding to the core sections (that are the central parts) of the the original captures. Both the size and position of this window, which we will refer to as *reduced window*, are algorithm parameters that have been derived from direct observation of common practices in websites and then refined through accuracy tests.

In Figure 3.5a we can observe the effects of the background banner on the full changesMap in which almost half of the blocks are evaluated as dynamic, in Figure 3.5b instead we present the same map observed through the reduced window and the corresponding sections of the original captures that are actually involved in computation. Thanks to the elimination of great part of the noise produced by the background banner we are able to classify these captures as representative elements of the same layout.



(a) Full size captures and full changesMap



(b) Captures and changesMap of Figure 3.5a observed through the reduced window.

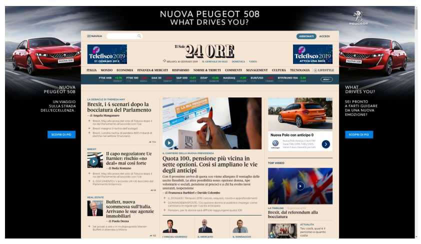
Figure 3.5: Captures classified exploiting Core Level Similarity

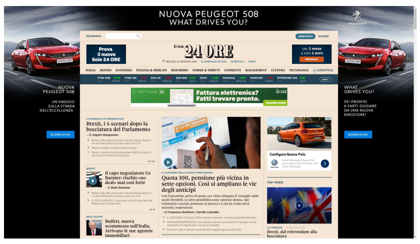
3.3.3 Pattern search

The approaches presented so far are very effective in many cases, in particular: the high level similarity measure is effective on those pages in which the dynamic contents counts for a limited percentage of the whole content rendered in the capture, on the other hand the core level similarity can handle effectively even those pages in which we have mainly dynamic contents by focusing on the core of the page. Yet both methods are not reliable in those cases in which we have many dynamic contents which are strictly interleaved with the static sections of the page occupying even the core of the page. To overcome this problem a completely different approach was needed. The search pattern routine (line 15 of listing 3.6) evaluates once again the changesMap derived from the comparison of two captures but this time we do not limit to the computation of the percentage of static and dynamic blocks rather we look for some particular patterns within the map itself. As done for high level and core level similarity routines, to enable this further step, we need to extend our base of funding principles with a new consideration:

(*iv*) In the vast majority of cases when a new dynamic content is added to a certain layout A producing a new layout B, the effect of this addition is to translate the whole content (both static and dynamic) laying in the area under the newly added dynamic content.

This means that comparing two captures belonging to layout A and B we will find that the new content itself and the whole area laying under it *up to the bottom of the page* is evaluated as dynamic. It is possible to get a visual idea of what stated in ***iv*** by looking at Figure 3.6 in which we can observe (highlighted in blue) the effects on the changesMap of the a small green banner that differentiates the layout in Figure 3.6a from the in Figure 3.6b.


(a) Layout A.


(b) Layout B.

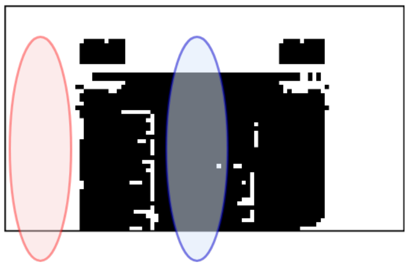

(c) Changes map.

Figure 3.6: Graphic evidence of consideration ***iv***.

Starting from ***iv*** it is possible to define typical patterns in changesMaps that only captures with the same layout can produce when compared. In particular if analysing the changeMap column by column from top to bottom we find a number of consecutive columns that end with a sequence of static blocks, we are sure that in that specific area no banner has been inserted. Of course, as we can easily observe in the red coloured area of the changesMap presented in Figure 3.6c, a set of columns terminating with a sequence of static blocks can be due to a common background or to sections of the page that are not interested by the translation, for this reason we refine our pattern research by:

1. ***Focusing on the core of the page*** : exploiting a reduced window to analyse our changes map we can once again reduce the noise due to the page background.

2. ***Targeting a more specific column pattern*** : we target a pattern including a sequence of dynamic blocks, followed by a sequence of static blocks up to the end of the page. This specific sequence provides a lot of informations, we get to know that: the slice of the evaluated captures corresponding to this column included both static and dynamic content, the dynamic content has changed but the static part has not been translated. This suggests that the evaluated area presents horizontal banners which are typical of the core sections of the page.

3. ***Looking for relatively long sequence of consecutive columns matching the pattern*** : this allows to be more confident that the evaluated area contains actual static contents as for example text boxes which tend to be larger with respect to vertical banners.

Following these principles the routine exploits once again the mechanism of the reduced window to focus on the core section of the page. Once we apply the window, we look at the resulting changesMap (which once again is an array of boolean values) as if it was a two-dimensional matrix representing that selected area, then we evaluate this matrix column by column from left to right and from top to bottom looking for the *Same Layout Pattern* which is described in listing 3.3.3.

```
((D | S)* D{ncd,} S{ncs,} E){ncc,}
```

The searched pattern is written in the form of a regular expression in which:

- **D**, **S** - represent the occurrence within a column of a dynamic or static;
- **E** - represents the end of a column;
- **ncd**, **ncs**, **ncc** - are respectively the algorithm parameters *Number of Consecutive Dynamic*, *Number Of Consecutive Static*, *Number of Consecutive Columns*;

In Figure 3.7 we see two captures classified through the search pattern routine. The area highlighted with the red rectangle is the section of the changesMap matching the *Same Layout Pattern*, the sequence of NCD dynamic and NCS static blocks are respectively signaled by the red and blue curly brackets while the sequence of NCC columns sis highlighted in green. We found the described method to be reliable and effective but, of course, errors are possible due to misjudged blocks or to particularly complex and chaotic of layouts and banners, thus in order to make the approach as robust as possible we assign a capture to a a certain cluster layout only if we identify the searched pattern in multiple reps of that layout. We count how many of the reps generate comparison with the searched pattern, this value is called *layoutConfidence* and each unassigned capture keeps such a value for each known layout. When the *layoutConfidence* with respect to a certain cluster exceeds a certain algorithm threshold called *layoutConfidenceTh* we assign the capture to that layout. The *layoutConfidenceTh* is expressed both as a percentage of current layout reps (for clusters with few eleemnts) and as numeric value of required matching reps (for bigger clusters), this duality is needed to allow assignments even to small clusters with few representative elements and to speed up computation in case of large layouts with tens of reps.

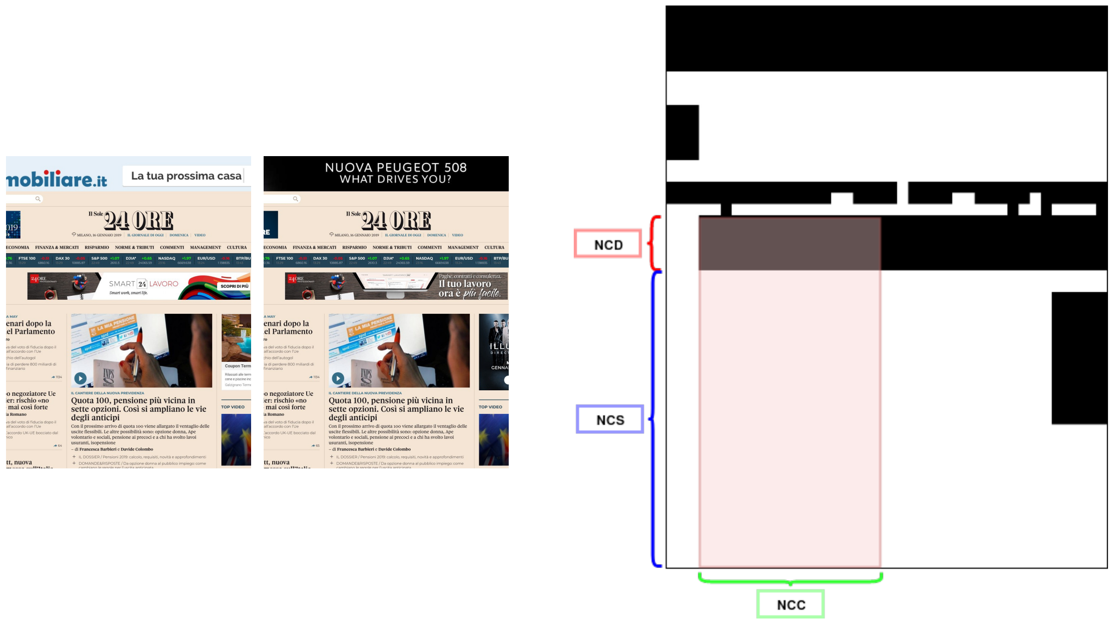


Figure 3.7: Captures classified using pattern search routine.

3.3.4 The low level banner problem

The search pattern routine can be considered the most crucial part of the algorithm as it enables clustering our captures even in those cases in which they are full of dynamic contents strictly interleaved with the static parts. As we previously said it is based on the consideration expressed in ***iv*** in which a main assumption is performed: we assume that if the blocks at the bottom of the capture are evaluated as static, then the page content above those block has not been translated. Despite in the vast majority of cases this assumption is valid, it is not always true that what we see at the bottom of a rendered page moves when the static page content is translated, or at least this is not true when evaluating captures from that page. The reason for this is immediately clear looking at Figure 3.8 in which three captures are showed that are all rendering the same main page layout, we can observe that the first two also render a banner covering a good part of the main page lower section. This kind of banners hides completely the informations that the search pattern routine exploits to perform its task.

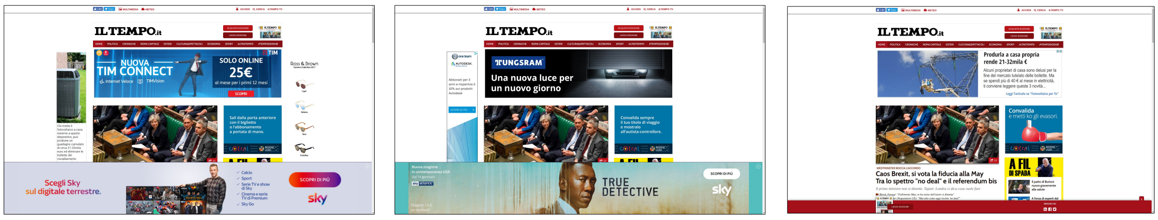

ILTEMPO.it
NUOVA TIM CONNECT
SOLO ONLINE 25€
Scegli Sky sul digitale terrestre.
SCOPRI DI PIÙ
sky
ILTEMPO.it
TUNGSRAM
Una nuova luce per un nuovo giorno
TRUE DETECTIVE
sky
ILTEMPO.it
Produrla a casa propria rende 21-32mila €
Caos Brexit, si vota la fiducia alla May
A FIL DI SPADA

Figure 3.8: Example of a web page which can render low level banners.

Being crucial for the developed approach to know where the *visible* bottom of the page is located, the low level banners constitute a problems that we need to address before even starting the layout detection task, we need to prepend to the layout detection phase a preprocessing phase. In particular this phase is actuated immediately after the captures have been converted into arrays of blocks. Examining the problem we see that it can be split in three different sub-problems:

1. ***Evaluate if a low level banner exists*** : this means we need to understand how many captures in our initial *capturesPool* are rendering a low level banner and then we also have to decide if the number of captures involved is big enough to take this aspect into account when performing the layout detection task.

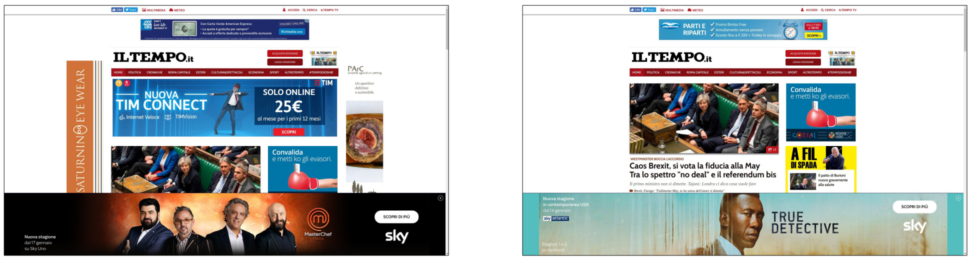
2. ***Estimate the dimensions of this banner*** : we need a reliable estimation of the eventual banner height and width in order to know which sections of the original page it is covering.

3. ***Calibrate the algorithm to avoid the banner*** : this can be done by simply rearranging the *changesMap* produced at line 7 of listing 3.6 using a proper window depending on the banner dimensions, the window can be defined in such a way to cut out from computation the section of the page occupied by the low level banner.

Once again also in this case to address the problem is crucial to exploit the knowledge related to the web domain in order to simplify our task. For this reason we need once again to extend our base of considerations with:

(*v*) In the vast majority of cases, the low level banners tend to be as wide as the whole page rendered.

Despite seeming trivial, the consideration in ***v*** allows deriving crucial hints in this context: first of all, we know that being these banners as wide as the adopted view port, the problem of estimating their size can be reduced to that of evaluating their height. The second important hint is related to another particular type of pattern that this types of banner can produce in the changesMap obtained comparing captures containing them. To get evidence of this new pattern we can take


(a) Compared images

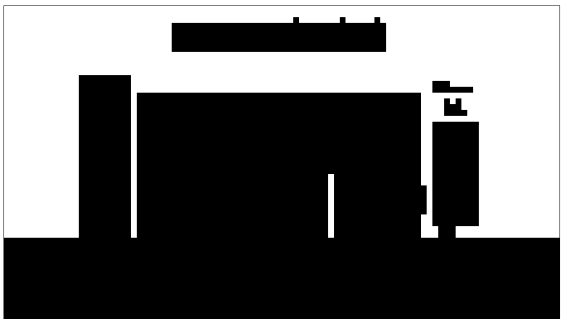

(b) Map of changed blocks.

Figure 3.9: Example of comparison among two captures showing low level banners.

a look at Figure 3.9 in which the changesMap derived from the comparison of two captures, implementing the same page layout and rendering low level banners is presented. The presence of large, screen wide, banners produces a uniform area of black coloured blocks in the bottom part of the changesMap, this area is placed exactly in the same position occupied by the low banners of the compared captures. The same uniform area is obtained any time we compare two captures rendering different low level banners and also when comparing a "normal" capture with one showing a low banner.

Given these evidences we have enough knowledge to drive the preprocessing algorithm. We need to search for the pattern produced by the screen wide black coloured area, by evaluating the height of this area we will be able to estimate the height of the rendered banners.The algorithm developed for this task can be summarized by the pseudocode in listing 3.7.

At the beginning of the computation a full mesh of comparisons is performed among all the captures and the obtained *changesMap* are stored in a list. The algorithm functioning principle is simple: we evaluate all the *changesMap* looking for a pattern suggesting the existence of a low level banner and its eventual height and take a decision exploiting a sort of voting system. The height of a banner is always measured in terms of number of blocks, thus its possible values range from 0 to *viewPortHeight* / *blockHeight*, for each of these possible height values a counter is maintained in the *bannerHeightVotes* dictionary. As soon we identify a banner and its possible height from a changesMap we increment a counter associated with that particular value of height, *we "vote" for that height.* On the contrary if no banner is found we do not increment any counter thus *we do not "vote".* The final decision is taken counting how many votes have been collected, if the total number of votes is greater than a certain threshold, we assume that a consistent part of our captures shows a low level banner, and its height is estimated to be the most voted value in *bannerHeightVotes*.

```
# Initializion steps
changesMapsList = GetFullComparisonMesh(capturesPool)
bannerHeightsVotes =
  InitHeightsVotes(viewPortHeight, blockHeight)

for( changesMap in changesMapsList ) {
  potentialBannerHeight = 0

  # List of map's rows in bottom to top order
  rowsList = ExtractMapRowsList( changesMap )

  for( row in rowsList ){
    if ( isRowDynamic( row ) )
      potentialBannerHeight += 1
    else
      break
  }

  if ( potentialBannerHeight > minBannerHeight )
    # We increment a counter associated to the found height
   value
    addVote(bannerHeightsVotes, potentialBannerHeight)

}
# Taking a final decision on banner existence and its height
bannerFound, estimatedHeight =
  evaluateVotes(bannerHeightsVotes, changesMapsList.len())
```

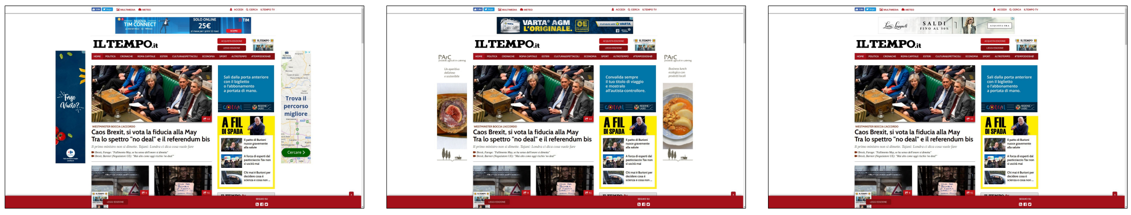
Listing 3.7: Low level banner identification algorithm pseudocode.

As said the pattern we look for in order to state the banner existence and its height is a very simple one. First we evaluate each map row by row from bottom to top, we compute for each row the percentage of dynamic blocks, if this percentage is higher than a given threshold the row is evaluated to be dynamic. Setting the value of this threshold we can decide the amount of dynamic blocks required to a row to be evaluated as dynamic, of course for the algorithm to be effective we have to be quite restrictive (applying thresholds higher than 0.9) but on the other hand we need to allow some static (white) blocks in dynamic rows to take into account *changesMap* deriving from partially similar low level banners. For each changesMap we count the amount of consecutive dynamic rows, this number is the *potentialBannerHeight*, if this value is higher than the *minBannerHeight* parameter we suppose that a low level banner has been identified in the capture and we add a vote for the *potentialBannerHeight* found.

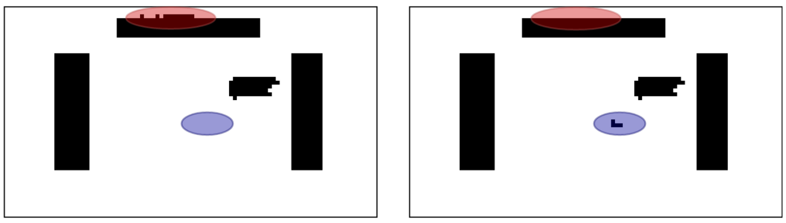
3.4 Core Sections Detection Algorithm

Once we get to group the captures within different layouts we select one of the clusters to be exploited for the subsequent step: the core sections detection. The algorithm has a quite simple workflow and is based on a single reasonable and simple assumption:

> (*i*) Given a set of captures showing the same template of a certain web page, if the captures have been collected within a reasonably limited amount of time, there is an high probability that comparing two of them the only differences can be detected within those sections showing the non relevant, dynamic content of the page.



(a) (b) (c)



(d) Changes map from *a* and *b* (on the left) and from *a* and *c*.

Figure 3.10: Differences in dynamic sections detected comparing different captures.

Starting from ***i*** we develop an effective approach to detect dynamic and non relevant page contents. The idea is simple: we compare our captures and identify the differing section, the system used for the comparisons is identical to the one used in section 3.3 with the main modification being the size of blocks which is reduced in order to obtain more fine grained maps describing dynamic and static sections. Despite the relative simplicity of the task, it is important to keep in mind that the assumption in ***i*** is valid as long as we take into account the term *probability*. In fact due to partial similarities among advertising contents or to rendering lags in in the static ones it is not always possible to correctly detect the dynamic sections by comparing only few captures.

In Figure 3.10 we can see the results of the comparison among captures implementing the same layout. Despite having the same page structure and thus being the positioning of the dynamic content identical in all of them, there are still small sections that the algorithm fails to identify properly. This problem has be addressed in two different ways:

1. ***Refining the analysis granularity*** : that is reducing the size of the blocks in which captures are split and that are the real objects compared. This allows to limit dynamic sections detection failures to smaller areas with respect to the big blocks used in Figure 3.10 (which are as big as the one used in the layout detection phase). Moreover in this way we also can identify with a greater resolution the perimeters of the areas of interest and notice even small differences. On the other hand this means that slightly similar dynamic contents will have more blocks classified as static contents.

2. ***Maximizing the evaluated comparisons*** : we need to perform a trade-off between the computation time and the amount of comparison evaluated. Examining a good number of captures is crucial to eliminate the noise caused by the small variations visible in Figure 3.10. For this reason we select the cluster with the greatest number of captures from the layout detection step.

The algorithm workflow can be described through the pseudocode presented in listing 3.8. The idea is that of performing a full mesh of comparisons among the captures, in other words each available captures is compared with all the other ones, where the comparison operation is based on the corresponding blocks subtraction and the non zero valued pixel counting operations already discussed in the layout detection algorithm. Despite this similarity in this case the result of each comparison is not a changesMap but rather an array of integers, each integer corresponding to the number of non zero valued pixels found for a certain block after the subtraction operation, these integers are called *NN0counts* and each comparison produces an *NN0count* value for each block. After each comparison operation we update the *totalPerBlockNN0Counts* (line 16 in listing 3.8) which is an object maintaining for each block a unique list of *NN0counts* which represent the results obtained for that specific block over the comparisons performed so far.

After all the full mesh of comparisons has been performed, we obtain for each block an array of NN0counts. This array is processed in the *evaluateResults* routine which takes a final decision over each block nature and is thus able to produce a changesMap collecting these decisions. Finally based on the changes map a visual result similar to the ones observed so far is computed and stored. The decision is taken keeping into account two thresholds:

NN0th : has the same nature and function of the threshold used in the comparison operations of the layout detection phase. It is the maximum percentage on non zero valued pixels allowed for a block to be considered a static one in the context of a single comparison.

PDEth : is the *Percentage of Dynamic Evaluations*. We count down how many times a block has been evaluated dynamic in the context of a single comparison over the total number of comparison obtaining its *blockPDE* value, if this value is greater than the *PDEth* than the block is definitively evaluated as a dynamic one.

```
# Initializion steps
splitCapturesList =
  SplitCaptures(capturesPool, blockHeight, blockWidth)

totalPerBlockNN0Counts =
  InitNN0Counts(len(splitCapturesList[0]))

while( len(splitCapturesList) >= 1) {
  # Get a new baseImg from list
  baseImg = splitCapturesList.pop()

  # Compare the base image with all the other captures
  for( img in splitCapturesList ) {
    imgNN0Counts =
      performComparison(baseImg, splitCapturesList)
    updateResults( totalPerBlockNN0Counts, imgNN0Counts)
  }
}

# Taking a final decision for each block
changesMap =
  evaluateResults(totalPerBlockNN0Counts, NN0th, PDEth)
computeAndSaveVisualResult( changesMap )
```

Listing 3.8: Low level banner identification algorithm pseudocode.

The values of *PDEth* and *NN0th* are crucial for the algorithm effectiveness as they are the delimiters distinguishing the noise due to rendering lags or to advertising similarities from the actual data, they have been refined through tens of experiments involving hundreds of captures within the local test environment that we will further discuss in chapter 4.

Chapter 4

Data Collection and Tests Setup

In this chapter the environments exploited in order to evaluate the effectiveness and performances of the algorithms presented in chapter 3 is introduced. First in section 4.1 we discuss the data collection phase and the shrewdness adopted in order to obtain sets of screenshots which could actually be used for our purposes. Then in section 4.2 we present the test environment created in order to evaluate the performances of the layout detection algorithm. Following in section 4.3 we go through the system developed in order to evaluate the section detection algorithm and to refine its parameters. Finally in section 4.4 we provide a detailed description of the two different dataset used for algorithm evaluation and in section 4.5 we summarize the suite of tests performed.

4.1 Data Collection Setup

The main concern during the data collection phase has been that of producing captures with some peculiar characteristics enabling their usage in the context of the algorithm tuning and testing, the "good capture traits" could be summarized by the following points:

1. **Coherent with the page content** : in simple words we want our captures to show the actual aspect of the visited page without the noise due to rendering lags or to page covering pop-ups and banners.

2. **Varied in dynamic contents** : for our approach to be effective we need the dynamic contents to change as much as possible among different screenshots of the same page in order to be able to detect these changes.

3. **Stable in static contents** : once again it is crucial for the developed solution to avoid captures collections in which the static, core contents of the page

changes as this would invalidate a good part of the considerations which the algorithms are base on.

All these points have been kept into account during the data collection phase. In order to achieve page coherence the most important actions were the modifications of the original Browsertime source code aimed to avoid the accept-cookie banners as we have already discussed in subsection 3.2.2, moreover the delay introduced between the page load event and the screenshot capture operation allows to reduce the probability of visible rendering lags in the captures.

To achieve point two of our good capture characteristics instead, we needed a mechanism able to influence the type of dynamic contents that we get to be rendered at each visit. To obtain this kind of behaviour, the most effective approach is that of exploiting the same trackers that we mentioned among the motivations of our work. Resuming what we have already discussed in section 1.2, we know that the contents rendered on the pages that we visit are strongly influenced by our previous interactions and in particular by our browsing history. Starting from this assumption we decided to create different chrome profiles targeted on specific themes of interest. We exploited the resources of the SimilarWeb[1] platform to identify the most relevant websites in five main areas: *Movies*, *FoodAndDrink*, *Sport*, *HealthAndFitness* and *Shopping*. We created a new chrome profile for each of these areas of interests and used it to visit the top twenty most relevant websites for that kind of contents according to SimilarWeb.

In this way, the profiles created have been made interact with a specific type of contents and thus, because of (or thanks to) the tracking services active on the visited websites, each of them has been profiled according to its interests. In this way we know that, visiting one of the websites in our test cases using all the targeted chrome profiles, we have a good probability of obtaining differentiated dynamic contents addressed to interests in different domains.

Finally to achieve the stability of the static contents of point three of the good capture characteristics, we simply had to minimize the delay among the first and the last visit to the targeted website. In the vast majority of cases we found that a delay in the order of minutes and even tens of minutes among visits does not produce any difference in the static contents rendered. To achieve this magnitude of delay we simply had to exploit multiprocessing in order to perform multiple visits, using different chrome profiles at the same time and by mean of different instances of Browsertime and then collect the obtained results in a single group of captures.

[1]SimilarWeb is a web platform that focuses its core business in providing global and multi device insights related to the digital market, for further details visit `https://www.similarweb.com/corp/about/`

```
1  #!/usr/bin/env bash
2  # Parameters <url> <perProfileVisits> <usrDataDir>
3  url=$1
4  perProfileVisits=$2
5  chromeProfileDir=$3
6  profileName=`echo ${3} | cut -d'/' -f3`
7  domain=`echo ${1} | cut -d '/' -f3 | cut -d '.' -f2`
8
9  dockerOptions=(
10 # Amount of memory shared by container process
11 '--shm-size=1g'
12 # Automatically clean up the container and remove
13 # the file system when the container exits
14 '--rm'
15 # Limiting resource usage: RAM 8gb CPUS 2
16 '--memory=8000m'
17 '--cpus=2'
18 )
19
20 browsertimeOptions=(
21 # Chrome is the default Browser
22 # We are only interested in screenshots
23 '--screenshot'
24 # Disabling video stuff
25 '--video=false'
26 '--visualMetrics=false'
27 # Cross device cookies
28 '--chrome.args=--password-store=basic'
29 )
30
31 echo "Running instance for profile ${3} on ${domain}"
32
33 docker run "${dockerOptions[@]}" \
34   -v "$(pwd)"/results:/browsertime-results browsertimeCustomBase \
35   "${url}" -n $perProfileVisits\
36   "${browsertimeOptions[@]}" \
37   --resultDir="browsertime-results/${domain}_${profileName}"
38   --chrome.args=--user-data-dir=/profile/${3}
```

Listing 4.1: Data collection script.

The data collection script is showed in listing 4.1. The docker options are exploited to limit the amount of resources used by the container and to persist the results through the mechanism of docker volumes (line 34). Among the Browsertime options provided, the most important ones are the parameters to be passed to the chrome instance (*–chrome-args*). The *–user-data-dir* argument specifies the location in the container filesystem of the chrome profile to be used for a certain set of visits while the *–password-store=basic* enables the docker chrome instance to read the profile cookie data even if they were produced by another chrome instance on another device as it forces the browser to store them in plain text.

4.2 Layout Detection Testbed

(a) corriere.it

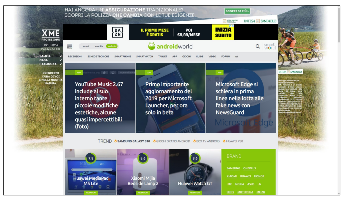
(b) hwupgrade.it

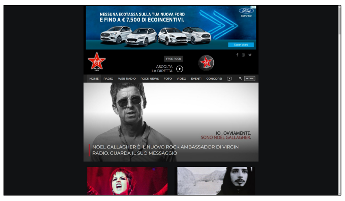
(c) aranzulla.it

(d) ilfattoquotidiano.it

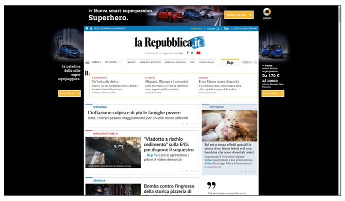
(e) meteo.it

(f) iltempo.it


(g) androidworld.it


(h) virginradio.it

(i) ilsole24ore.com


(j) repubblica.it

(k) gazzetta.it

Figure 4.1: Websites for the layout detection testbed.

The first test environment we had to set up was the one related to the layout detection task. We collected one hundred captures from each of the eleven websites of our test base, exploiting the data collection tool and taking advantage of the targeted chrome profiles and of multiprocessing to obtain sample presenting the characteristics of good captures. In Figure 4.1 we can see the full set of websites which provided the captures used in the layout detection tests.

Multiple guidelines have been kept into account in order to decide which websites had to be used: the main aspect to keep into consideration has been the variety of the test suite, variety was required both in page structure complexity and in layout differentiation. To achieve this variety, we included both sites with a very simple structure of the page as `aranzulla.it` (which, as we can observe in Figure 4.1, renders only a background and a side banner) and websites with multiple advertising and complex structure as `iltempo.it` (which can also render a low level banner). At the same time, the number of possible layouts rendered by each websites mirrors the variety of a real web environment in which we find both very simple sites as `meteo.it` which show less than three different layouts and more complex ones as `ilsole24ore.com` which instead implement more than seven possible page structures. These peculiarities lead to differences in terms of complexity and time requirements of the layout detection task for each website. Once we had collected our captures, we built the ground truth by simply grouping them by layout and storing the optimal grouping within a json file.

Along with the ground truth provided by the classified captures, an effective system to test and evaluate possible combinations of parameters was required. Both the developed algorithms accept in input a json file containing the parameters to be used for a specific algorithm run in the form of named properties. As we can observe in Appendix A, the amount of possible parameters combinations is significantly high for the layout detection algorithm. For this reason we defined a script able to automatically generate the configuration files to be provided to the algorithm, this script accepts in input a json file called *testArguments* which contains a description of which parameters have to be tested and in which range they must be varied. Given the *testArguments* file the script computes all possible resulting combinations of parameters and generates the related configuration files.

```
{
  "clusteringParams":{
    "blockHasChangedThreshold": [0.4, 0.51, 0.1],
    "highLevel_minDistance": [0.1, 0.251, 0.05],
    "highLevel_maxDistance": [0.8, 0.91, 0.1],
    "core_minDistance": 0.2,
    "core_maxDistance": 0.8
  }
}
```

Listing 4.2: Example of testArguments json file.

The *testArguments* file showed in listing 4.2 generates 16 named configuration files. The varying arguments are expressed as triples in the form [*minValue*, *maxValue*, *step*], instead the non varying ones are expressed as single values. In case some algorithm parameters are not even mentioned within a *testArguments* file they are set to default values.

Once we obtain the set of configuration files needed, we run another script which exploits multiprocessing to perform multiple runs of the algorithm with different configurations over the whole set of captures of our testbed. The results of each run are compared with the optimal layout grouping associated to each test case, the outcomes of these comparisons are collected for each website and for each named configuration. The evaluated metrics include measures of accuracy both in terms of correctly detected layouts and of correctly assigned captures as well as the computation time, moreover for each configuration the average accuracy and computation time measures are computed. In listing 4.3 we can observe a json file produced as output by a test run.

```json
{
"summary": {
    "cfg0": {
      "avgCorrectlyClassifiedImgs": 0.984,
      "avgCorrectlyClassifiedLyts": 0.937,
      "avgElapsedTime": 97.604
    },
    "cfg1": { ... },
    [...]
  },
  "cfg0": {
    "meteo": {
      "correctlyClassifiedImgs": 1.0,
      "correctlyClassifiedLyts": 1.0,
      "elapsedTime": 25.746
    },
    [...]
  },
  "cfg1": { ... },
  [...]
}
```

Listing 4.3: Example of test run results.

4.3 Core Sections Detection Testbed

The second test environment we had to build was the one for the core sections detection algorithm. The main problem with this specific test environment was that of creating a ground truth for the subsequent testing. In fact what is needed in order to evaluate the performances of the this particular algorithm is knowing the exact position of the static contents within the page in order to be able of distinguish which blocks (with the meaning of sub areas) of the tested captures have to be evaluated as static and which ones instead have to be marked as dynamic.

The approach adopted to solve the problem was that of creating a local environment able to simulate the rendering of a page in which the position and nature (static or dynamic) of each section could be known a priori. This environment has been developed in the form of a simple Angular 6[2] web application. This particular framework has been chosen because of its capability of enabling the fast development and immediate testing of modular and extensible applications. The developed solution has been called *PageGenerator*, it is capable of rendering pages with an arbitrary structure described by mean of configuration files in json format. In listing 4.4 we can observe a possible configuration file for the page generator.

```
envConfig:{
  MAXBLOCK_PER_ROW: 192,
  MAXBLOCK_PER_COLUMN: 108,
  baseBlockWidthPx: 10,
  baseBlockHeightPx: 10,
},
activeConfig: 'externalBands',
externalBands: {
  structure: [
  {xSpan: [0, 192], ySpan: [0, 16]},
  {xSpan: [0, 24], ySpan: [16, 42]},
  {xSpan: [24, 168], ySpan: [16, 26], secType:'static-title'},
  {xSpan: [168, 192], ySpan: [16, 42]},
  {xSpan: [0, 24], ySpan: [42, 68]},
  {xSpan:[24, 168], ySpan:[26, 32], secType:'static-sub-title'},
  {
    xSpan: [24, 168],ySpan: [32, 50],
    blockType:'static-img',
    resourceReference: '0.jpg'
  },
  {xSpan: [24, 168], ySpan: [50, 108], blockType:'static-text'}
 ]
}
```

Listing 4.4: Page generator configuration file example.

The structure of the page is defined as a grid (a matrix) of blocks, the parameters defining this grid are grouped in the *envConfig* property which defines the size in pixels of each basic block of the grid and the maximum number of blocks per row and per column (in this way the viewport of the browser window is defined as well). We can maintain multiple configuration and change the actually rendered page by simply modifying the *activeConfig* property. Each configuration includes a structure property which is an array defining the different sections to be rendered within the page. Sections have rectangular shape (as it is common in the vast majority of web pages), each of them declares two mandatory properties: *xSpan* and *ySpan* which are tuples describing the size and position of the section within the grid by providing respectively the starting and ending column and row indexes

[2]Angular is modern framework based on TypeScript for the development of web applications. See https://angular.io/.

of the blocks composing the section. According to these properties, each section is rendered on the page with fixed height and width, moreover its position within the page is fixed too. This behaviour is achieved by providing to each section, which is rendered as an HTML <div> element with variable content, custom CSS[3] properties computed at page load time according to the specific section structure.

We can explain the mechanism by mean of an example, we will examine the second structure defined in the *externalBands* configuration (line 10 of listing 4.4). The structure describes a section spanning horizontally from block 0 to block 24, this means that the corresponding <div> element will be (24 - 0)**baseBlockWidthPx* wide, the section horizontal positioning is given by its leftmost block index, in this case it is 0 thus the section is on the leftmost side of the page. In a similar way, looking at the *ySpan* property we can derive that the section height is equal to (42 - 16)**baseBlockHeightPx*, the vertical positioning instead is given by the topmost block index that is 16 thus the section will be placed at 16**baseBlockHeightPx* pixels from the top of the page. Given the *envConfig* in listing 4.4, the resulting CSS properties applied to the div generated for this section will be those showed in listing 4.5.

```
{
  width: 240px,
  height: 260px,
  position: absolute,
  left: 0,
  top: 160px
}
```

Listing 4.5: CSS properties computed for the second section of the *externalBands* configuration of listing 4.4.

Beside its shape and position, each page structure can be characterized by a *secType* property. Within the created environment, which is a simplification of the real web, we have defined six different types of contents for a generic section:

dynamic : dynamic sections are used to simulate the advertising content of a page, the corresponding <div> element will contain an advertising banner that is randomly selected for each section of this type at page load from a pool of one hundred advertising images. It is the default section type and is assigned to all those section defined in a configuration but lacking a *secType* property.

static-title : the corresponding <div> element will include an <h2> HTML tag with a predefined title.

[3]CSS stands for *Cascading Style Sheet*, it is a language describing the style (mainly graphics and animations) of an HTML document. See `https://www.w3schools.com/css/`.

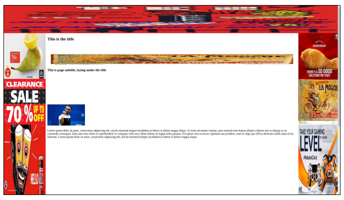
static-sub-title : the corresponding <div> element will include an <h3 > HTML tag with a predefined sub title.

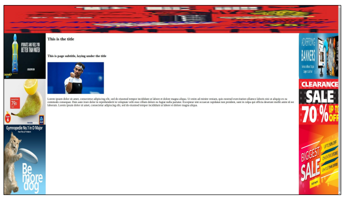
static-img : the corresponding <div> element will include a static image specified by the *resourceReference* property.

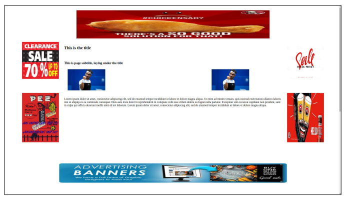
static-text : the corresponding <div> element will include a <p> HTML tag dynamically filled using a lorem ipsum [4] generator.

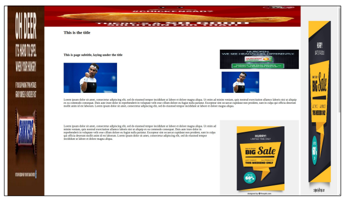
placeholder : all the grid blocks which are not part of the sections defined by a configuration are rendered as placeholder which are simple empty <div> elements with the exact dimensions of a grid block. This type of elements allow to have a visual feedback of the grid when exploring the page using the browser dev-tools.

As already said the configuration is processed at page load, and according to the structure elements, different page sections are generated, shaped and positioned through CSS properties. Moreover as we "build" the sections to be rendered, it is maintained track of which blocks of the grid are occupied by which type of content, this is done both in order to signal bad configurations containing overlapping section but above all to produce for each configuration an array of boolean stating the nature of each block of the rendered grid. This array is actually identical to a *changesMap* and thus can be used as ground truth for the test cases produced using the *PageGenerator* application.


(a) repubblicaLocal


(b) externalBands


(c) isaolatedBanners


(d) asymmetricBanners

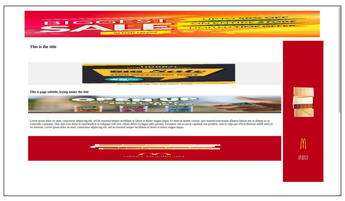

(e) mainlyDynamic

Figure 4.2: Pages generated for section detection testing.

[4]Lorem ipsum is a latin text often used as placeholder for text contents.

The test suite for the section detection algorithm has been built using the data collection tool to visit on our local machine the pages produced by five different configurations of the *PageGenerator* application (inspired to common page layouts from the web) and by collecting the related *changesMap* to be used as ground truth. In Figure 4.2 we can observe captures of the generated pages with their configuration code name.

The testing routine is similar to the one used for the layout detection algorithm, the first step is that of generating configuration files to feed the algorithm. In this specific case the testing routine compares the *changesMap* produced by the page generator with the one resulting from the algorithm run. The core section detection problem as said is observed as a classification task in which an object (a block and thus a sub section of the original capture) can be classified as static or dynamic, thus we compute *Accuracy*, *Precision*, *False Discovery Rate* and *True Positive Rate* which are typical measures for this kind of tasks.

4.4 Datasets

In this section we will provide a detailed description of the datasets used for testing the algorithms.

The dataset used for the layout detection algorithm testing is made of captures from ten real websites, as already said the number of layouts and thus the complexity of the classification task varies consistently among websites. We will refer to this dataset as **`ldWildDataset`**, it is detailed in Table 4.1 remarking for each test case the corresponding website, the total amount of captures collected for that site and the number of different layouts which can be identified.

The dataset used for the core sections detection algorithm testing instead is made of captures obtained from different configurations of the *Page Generator*. We will refer to this dataset as **`sdSynthDataset`**, it is detailed in Table 4.2 remarking the total amount of captures for each test case and the percentage of dynamic blocks over the total.

Codename	**Website**	**# Captures**	**# Layouts**
androidWorld	`www.androidworld.it`	100	7
meteo	`www.meteo.it`	100	2
sole24ore	`www.ilsole24ore.com`	100	9
repubblica	`www.repubblica.it`	100	3
corriere	`www.corriere.it`	100	10
hwUpgrade	`www.hwupgrade.it`	100	3

aranzulla	www.aranzulla.it	100	3
virginRadio	www.virginradio.it	100	3
gazzetta	www.gazzetta.it	100	7
fattoQuotidiano	www.ilfattoquotidiano.it	100	7

Table 4.1: Details of **`ldWildDataset`**.

Codename	**# Captures**	**% Dynamic blocks**
lolaRep	100	40.10
externalBands	100	36.11
isolatedBanners	100	25.31
mainlyDynamic	100	46.72
asymmetricBanners	100	42.04

Table 4.2: Details of **`sdSynthDataset`**.

4.5 Test suites

In this section we will provide an overview of the test suites for the developed algorithms remarking for each test the involved parameters, the values they assume and the amount of tested configurations. We will refer to the names of the parameters as described in Appendix A. In Table 4.3 we group the tests related to the layout detection algorithm, while the test suite of the section detection algorithm is described in Table 4.4. In both table we adopt the notation [*startValue*, *finalValue*, *step*] to describe parameters ranging within a certain interval instead we will write <val1, val2, val3, ..> to describe parameters assuming a set of values.

4.5.1 Layout Detection test suite

Test name	Tested Params	Range	# Config.
t01blockSize	blockWidth blockHeight	<5, 10, 15, 20, 30, 40> -[5]	6
t02nn0Th	NN0Th	[0.05, 0.35, 0.01]	30
t03hlMinDist	highLvlMinDist	[0.05, 0.33, 0.02]	15
t04hlMaxDist	highLvlMaxDist	[0.65, 0.93, 0.02]	15
t05crWdwWidth	coreWindowXStart coreWindowXSpan	[20, 31, 1] [36, 52, 4]	12 x 5
t06crWdwHeight	coreWindowYStart coreWindowYSpan	[0, 9, 1] [44, 54, 2]	36 [6]
t07clMinDist	coreLvlMinDist	[0.15, 0.43, 0.02]	15
t08clMaxDist	coreLvlMaxDist	[0.55, 0.83, 0.02]	15
t09ptrnConsec	patternNCC	[2, 20, 1]	19
t10ptrnShape	patternNCD patternNCS	[0, 6, 2] [2, 14, 1]	12 x 4
t11ptrnConsShape	patternNCC patternNCS	[6, 12, 1] [9, 19, 1]	7 x 11
t12srcWdwWidth	srchWindowXStart srchWindowXSpan	[7, 11, 1] [10, 17, 1]	5 x 8
t13srcWdwHeight	srchWindowYStart srchWindowYSpan	[0, 4, 1] [48, 54, 1]	25 [7]

Table 4.3: Test suite for the layout detection algorithm.

[5]We use squared blocks thus width and height assume always the same value.

[6]Manually provided configurations as not all values are legal for each algorithm configuration.

[7]Similar considerations with respect to *t06crWdwHeight*.

4.5.2 Core Sections Detection test suite

Test name	**Tested Params**	**Range**	**# Config.**
sdt1nn0pde	NN0Th PDE	[0.20, 0.84, 0.02] [0.6, 0.9, 0.1]	33 x 4
sdt1captureLimit	limitCaptures	[6, 100, 1]	95
sdt2captureLimitTime	limitCaptures	<50, 45, 40, 35, 30, 25, 20, 15, 10, 5>	10

Table 4.4: Test suite for the section detection algorithm.

4.6 Hardware configuration

Here we present a comprehensive description of the hardware used to run our test suites.

```
DeviceClass     Description
========================================================

processor       Intel(R) Core(TM) i5-2450M CPU @ 2.50GHz
memory          32KiB L1 cache
memory          256KiB L2 cache
memory          3MiB L3 cache
memory          8GiB System Memory
memory          4GiB SODIMM DDR3 Synch. 1333 MHz (0,8 ns)
memory          4GiB SODIMM DDR3 Synch. 1333 MHz (0,8 ns)
display         GF119M [GeForce 610M]
network         Centrino Wireless-N 100
network         AR8151 v2.0 Gigabit Ethernet
disk            500GB WDC WD5000LPLX-0
```

Chapter 5

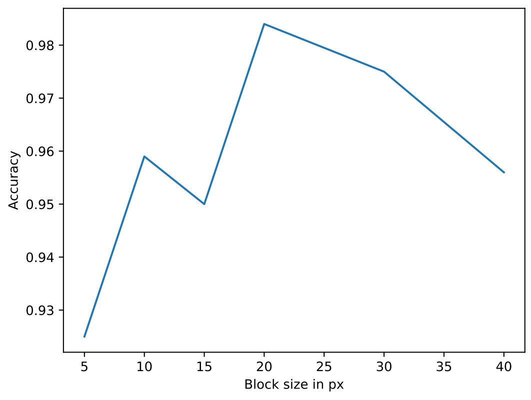
Results

In this chapter we analyse the results of the testing sessions performed on the developed algorithms, we go through each test discussing the reasons behind each evaluation and highlighting the single steps of the algorithm refining process. First in section 5.1 we go through the parameter refining process of the layout detection algorithm in which, given the amount of possible setups, a result driven approach has been adopted meaning that each set of parameters has been approached as an independent sub problem. Then in section 5.2 we present the results obtained testing the core sections detection algorithm in which, thanks to a minor amount of parameters to be kept into account, a more extensive testing approach has been adopted. Finally in section 5.3 we provide a summary of the outstanding results achieved both in terms of accuracy and time performances by the algorithms. During the results presentation and the test explanation the reference of Appendix A could be useful to help following the various step by allowing to rapidly and schematically resume the various parameters names and their meaning, Table 4.3 and Table 4.4 instead can be exploited to associate the name of each test to its actual scope.

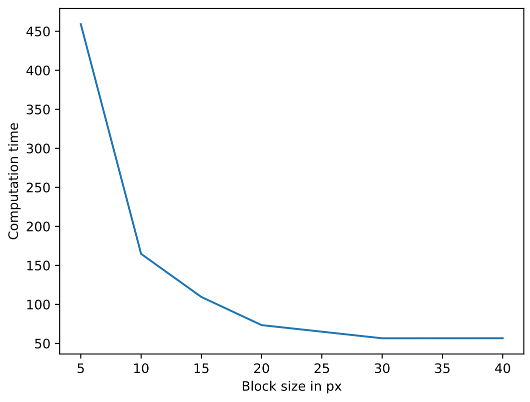
5.1 Layout Detection parameters tuning

The definition of the optimal configuration for the layout detection algorithm required a step by step analysis, the amount of parameters and thus of possible configurations required multiple specific tests aimed at tuning single or couples of parameters. Each test result has been analysed considering the necessity of a trade-off among computation time and accuracy in layout detection.

The first high level parameters to be set were those related to the most basic unit of information analysed by our system, the sub blocks in which each capture is divided. We decided to use squared blocks in order to approximate a punctual analysis of captures. Taking a decision about the dimensions of these squares required performances and accuracy evaluations: using large blocks allows to reduce the computation time as we need to process grids with less elements, on the other hand increasing block dimensions means coarser grained analysis and thus higher probability of errors.



(a) Average percentage of correctly assigned captures.



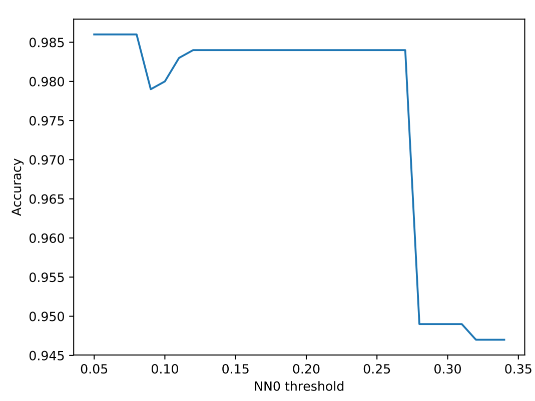
(b) Average time elapsed for classification.

Figure 5.1: Performance metrics for different block sizes.

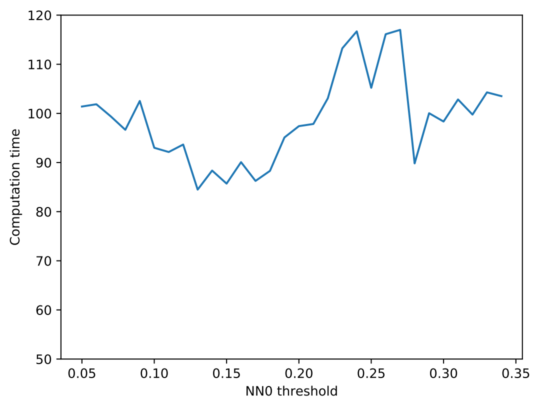
In Figure 5.1 we can observe the results of the *t01blockSize* test described in Table 4.3, in particular in Figure 5.1a the trend of the accuracy metric (measured as the average percentage of captures assigned to the correct layout in our test cases) when varying the block dimensions is showed. As we can notice accuracy decreases for very small block sizes (under 10px), this is due to the nature of the *searchPattern* routine which looks for uniform and large patterns of static and dynamic blocks, using small blocks we increase the sensitivity of our analysis and thus we are more sensitive to the noise produced by small similarities among banners or page rendering lags, for this reason it is harder to detect the searched pattern. On the other hand accuracy performances are also negatively affected by too large blocks which make the analysis too coarse to allow detection of small banners and minor differences among captures. The best result in terms of accuracy is obtained for squared blocks of side 20px.

As expected the average computation time instead decreases constantly as we increase block dimensions although the gains become of the order of seconds for sizes greater than 20px and even of tens of seconds for blocks of side greater than 30px. Given these results the block side dimensions has been set to 20px for all the subsequent tests.

The second logical step has been that of refining the *NN0th* threshold, a crucial parameter that is used to discriminate equal and differing sub areas of examined captures. Once again the value of this threshold is a measure of how sensitive we are in detecting changes among two blocks. In Figure 5.2a we can observe the results of *t02nn0Th* test from Table 4.3. It is remarkable to notice how gradually increasing this threshold we can produce a decreasing trend of the accuracy metric.



(a) Average percentage of correctly assigned captures.

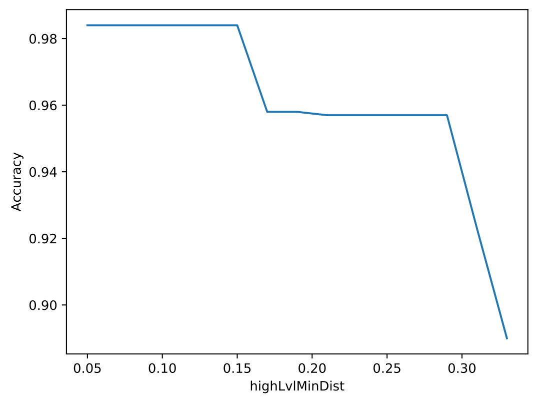


(b) Average time elapsed for classification.

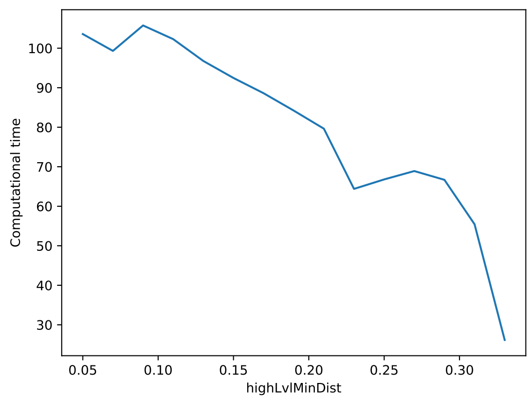
Figure 5.2: Performance metrics for different *NN0th* values.

Increasing the *NN0th* value means accepting to recognize as not changing (static) sections blocks that differ for a growing amount of pixels, this implies an higher probability of errors. On the other hand being slightly less strict allows avoid detecting non relevant differences among blocks due to small rendering lags and this means improving time performances of the classification task as showed in Figure 5.2b.

At this stage we have an optimal value for the block size and the *nn0Th*, we start focusing on tuning the parameters of the classification routines. As previously said, the *highLvlSimilarity* routine relies on two parameters whose role is summarized in Appendix A: *highLvlMinDist* has been evaluated in *t03hlMinDist*, while *highLvlMaxDist* has been tested in *t04hlMaxDist*.



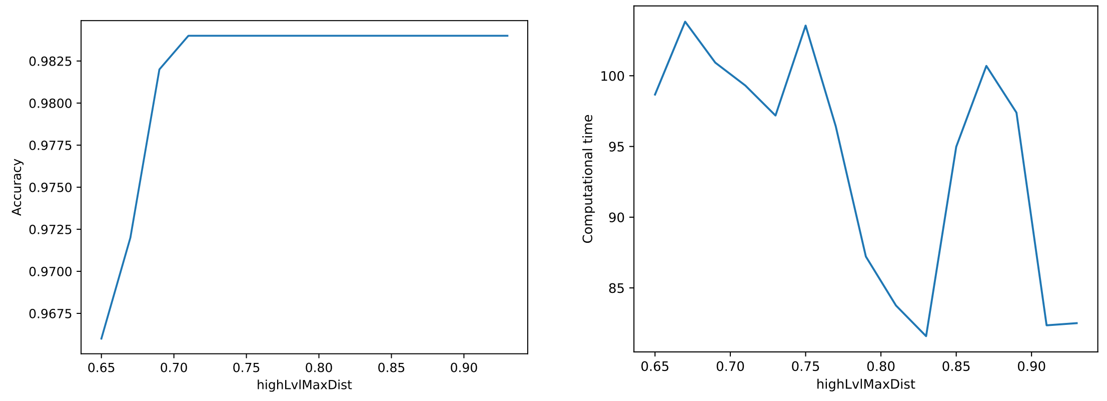
(a) Average percentage of correctly assigned captures.



(b) Average time elapsed for classification.

Figure 5.3: Performance metrics for different *highLvlMinDist* values.

The results of *t03hlMinDist* are presented in Figure 5.3. The resulting trend is quite evident, setting too high values for this threshold causes losses in accuracy, it is important to keep this threshold as strict as possible, in fact observing the whole capture it is not unusual to come across tricky captures belonging to different layouts but showing a not so high value for the high level distance measure, this could the case of two layouts implementing a background banner and two captures belonging to different layouts showing the same exact background. Of course the computation time decreases if we accept to be less strict on this parameter as we are less strict in assigning captures to a layout and as a consequence we perform less comparisons.



(a) Average percentage of correctly assigned captures.

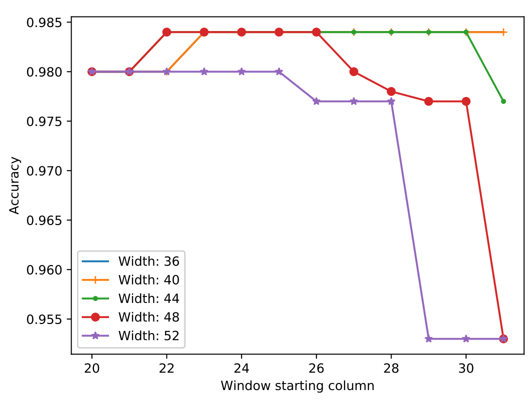
(b) Average time elapsed for classification.

Figure 5.4: Performance metrics for different *highLvlMaxDist* values.

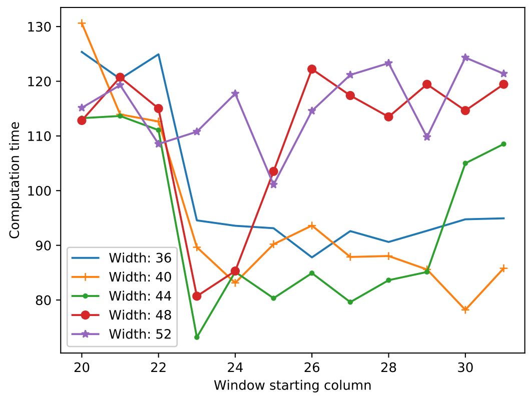
The results of *t04hlMaxDist* showed in Figure 5.4 can be commented with similar considerations. The accuracy decreases as we start considering as incompatible captures differing in less than 75% of blocks and this can generate errors for pages with mainly dynamic contents. Of course the computation time increases reducing the threshold as we tend to split into multiple groups captures from a single layout, more layouts of course means more comparisons needed and thus longer computation before understanding that a certain capture belongs to a new layout.

By setting up the *highLvlMaxDist* we define the algorithm behaviour up to the *High Level Similarity* routine thus we are ready to evaluate the parameters related to the *Core Level Similarity* routine. The first values to be set are the ones defining to the shape and position of the *CoreWindow*, this is logically the first thing to do as the performances of *coreLvlMinDist* and *coreLvlMaxDist* thresholds are strictly dependent on the used window. Given the shape and position of the core window, we identify a particular area of the analysed captures which, on average, will show across the majority of websites a certain composition in terms of percentage of static and dynamic blocks, the value of thresholds used in this phase has to be set according to the composition of the selected window.

We decided to split the analysis in two parts, setting horizontal position (the column index of the leftmost blocks of the window) and width in test *t05crWdwWidth* and taking care of vertical position (the row index of the topmost blocks of the window) and height in *t06crWdwHeight*.



(a) Average percentage of correctly assigned captures.



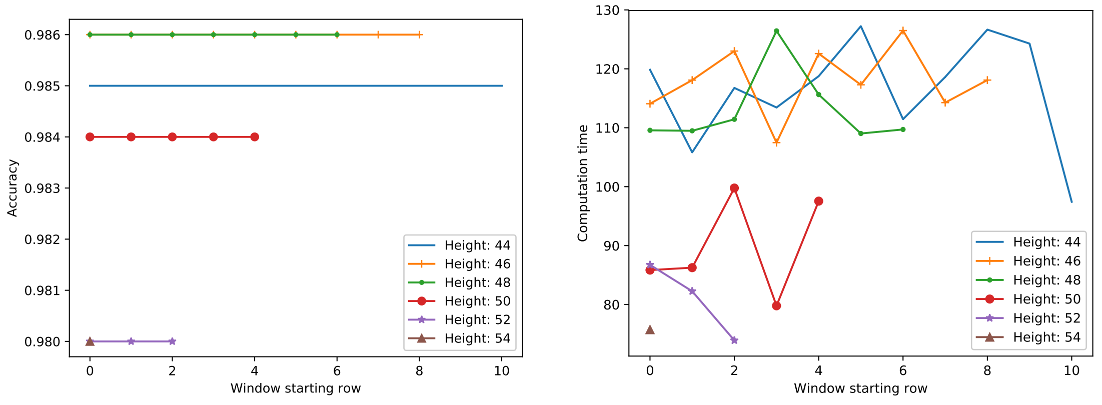
(b) Average time elapsed for classification.

Figure 5.5: Performance metrics for different *coreWindow* widths and horizontal positioning.

Observing the results of *t05crWdwWidth* in Figure 5.5 we can notice that, as expected, performances in terms of accuracy drop if we depart too much from the horizontal center of the capture. Despite this consideration we found that for *changesMap* containing 96 blocks per row the best performing configuration is not exactly at the center of the viewport but is slightly shifted to the left having a a width of 44 blocks and 23 as starting column index. This can be explained if we think that the aim of the core window is to focus on sections containing as much static (core) contents as possible in order to ease the discrimination among layouts by focusing on less variable elements: many websites implement a column containing feeds and small banners on the right side of the core content section thus it is not surprising that the best configuration partially cuts off such kind of elements to focus on those sections that usually tend to contain more relevant informations.

In Figure 5.6 we present the results of *t06crWdwHeight*. The most remarkable thing to notice is that the starting row index seems to have no effect on accuracy which instead is influenced only by the window height. The best performing combination was the one presenting a 46 blocks high window starting at row 3 of the *changesMap* that is 60 pixels under the top of the capture for our parameters settings. Moreover this height value excludes the bottommost section of the page which often can contain partially visible banners and contents that are difficult to analyse.

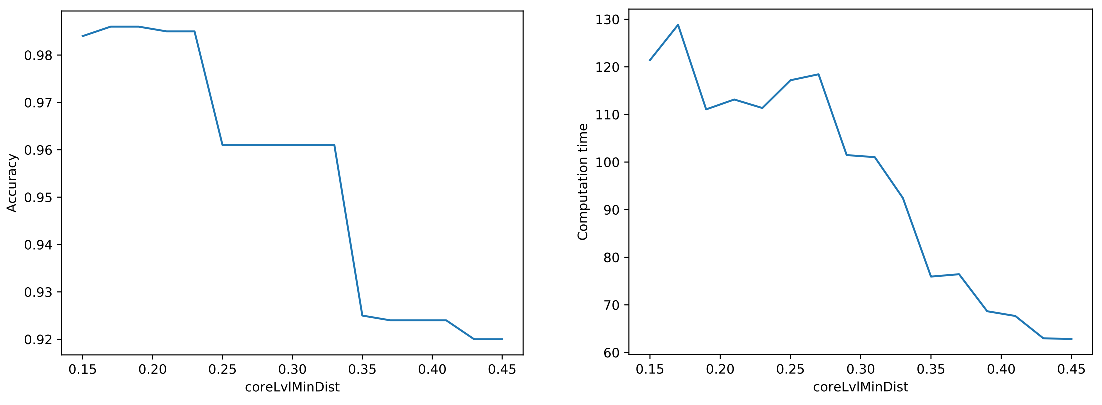
Once positioned and shaped the *coreWindow* we are able to set meaningful



(a) Average percentage of correctly assigned captures. (b) Average time elapsed for classification.

Figure 5.6: Performance metrics for different *coreWindow* heights and vertical positioning.

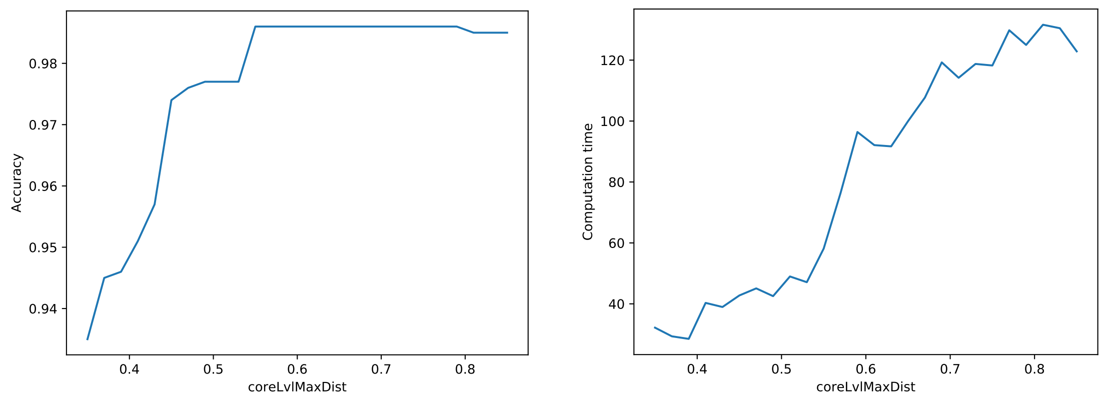
thresholds for the *Core Level Similarity* routine. In *t07clMinDist* and *t08clMaxDist* we tested multiple possible values of *coreLvlMinDist* and *coreLvlMaxDist*, tests results are presented respectively in Figure 5.7 and Figure 5.8.



(a) Average percentage of correctly assigned captures. (b) Average time elapsed for classification.

Figure 5.7: Performance metrics for different *coreLvlMinDist* parameter values.

Similarly to what we have observed for *highLvlMinDist*, increasing the value of the *coreLvlMinDist* makes the algorithm less strict in assigning a capture to a template for *Core Level Similarity* thus it is easier to commit errors and accuracy decreases, on the other hand we obtain a boost in computation time because we perform less comparisons. Regarding the *coreLvlMaxDist* parameter, decreasing its value our algorithm is more likely to declare captures layout incompatible for *Core Level Non Similarity* this meaning incrementing the probability of errors when observing captures with rich dynamic contents in core if we set it to excessively low values. By the way declaring incompatibility among layouts is very effective in speeding up computation as we are faster in discovering new layouts skipping useless comparisons among incompatible captures, setting wisely the *coreLvlMaxDist* we achieved great gains in terms of computation cost of the algorithm as we can observe in Figure 5.8b. The *coreLvlMaxDist* has been set to 0.55 in order to maximize accuracy still obtaining a great improvement in computation cost with respect to previous configurations (gain is in the order of tens of seconds with respect to the previous test results showed in Figure 5.7b).



(a) Average percentage of correctly assigned captures.

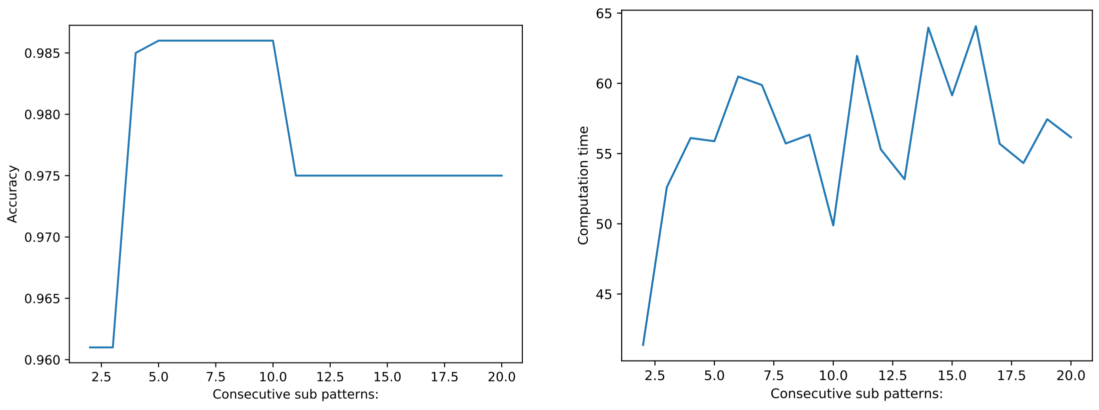
(b) Average time elapsed for classification.

Figure 5.8: Performance metrics for different *coreLvlMaxDist* parameter values.

Identifying the best configuration for the *Search Pattern Routine* parameters required some trial and error. The effectiveness of a certain *searchWindow* is strictly related to the shape of the pattern we look for which is defined by *patternNCC*, *patternNCS* and *patternNCD*; in this case it is also true the opposite statement: a particular pattern shape is convenient only for certain *searchWindow* configurations.

As first approach, we decided to keep the default values for all other parameters and look for an optimal value of *patternNCC* in *t09ptrnConsec*. The results can be evaluated in Figure 5.9. The *patternNCC* defines the width of the area in which we look for the sub-pattern of listing 3.3.3 thus of course it has to be set to a value smaller with respect to the average width of the core static content section of the average website, on the other hand it has to be enough large to avoid the possibility of grouping together captures showing the same side banner but differing in the main content layout.

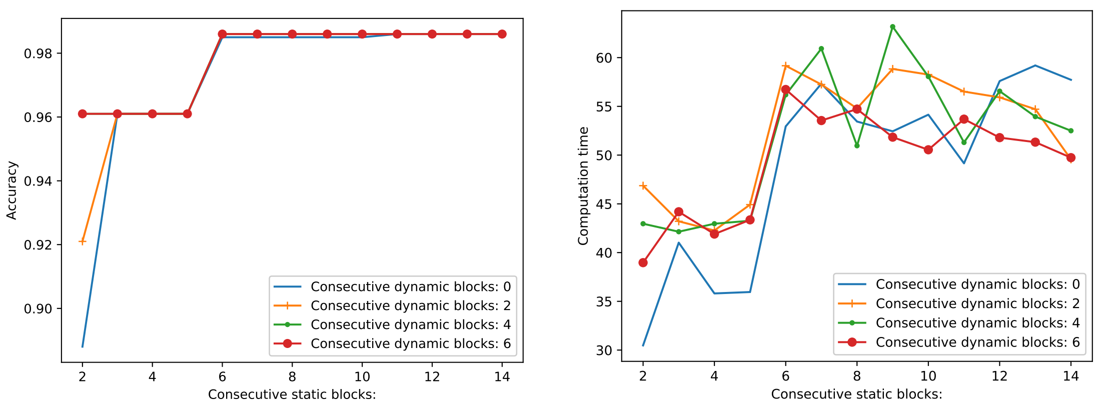
After this first attempt, without setting the optimal value found for *patternNCC* in *t10ptrnShape* we evaluated combinations of *patternNCS* and *patternNCD* which define the shape of the sub-pattern we look for. The results presented in Figure 5.10 showed how the influence of *patternNCD* is minimum in terms of accuracy. For this reason we decided to set a default value for *patternNCD* and evaluate combinations



(a) Average percentage of correctly assigned captures. (b) Average time elapsed for classification.

Figure 5.9: Performance metrics for different values of *patternNCC*.

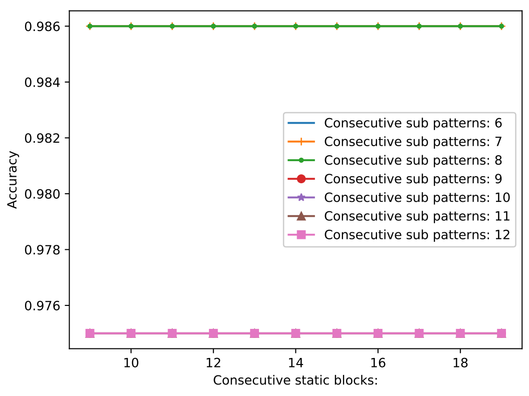
of *patternNCC* and *patternNCS* in *t11ptrnConsShape* whose results are presented in Figure 5.11. The test results in terms of accuracy highlighted once again how crucial the *patternNCC* parameter is. We decided to define the final pattern shape setting the parameters to the best performing configuration of *t11ptrnConsShape* with *patternNCC* equal to 7, *patternNCS* equal to 10 and *patternNCD* equal to the default value of 2.



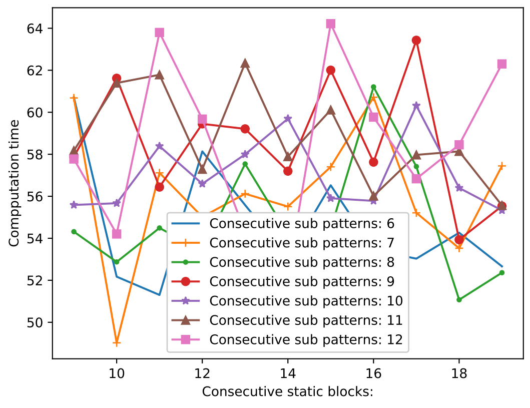
(a) Average percentage of correctly assigned captures. (b) Average time elapsed for classification.

Figure 5.10: Performance metrics for different combinations of *patternNCD* and *patternNCS*.

To complete the configuration of the layout detection algorithm we ran tests *t12srcWdwWidth* and *t13srcWdwHeight* to define the final position and shape of the *searchWindow*. The results of these final tests are showed respectively in Figure 5.12 and Figure 5.13. Regarding width and horizontal positioning we can notice in



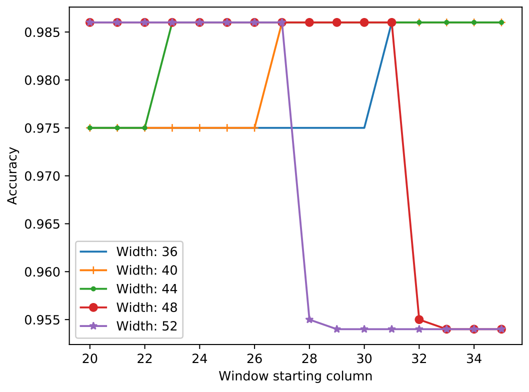
(a) Average percentage of correctly assigned captures.



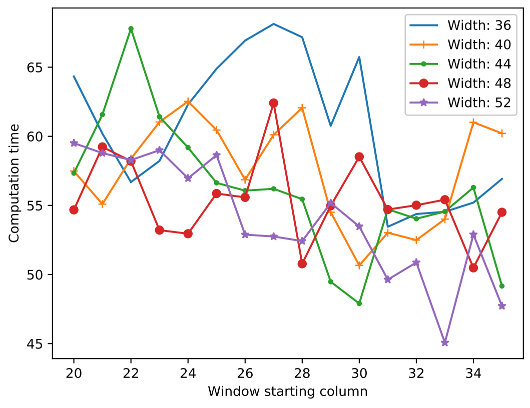
(b) Average time elapsed for classification.

Figure 5.11: Performance metrics for different combinations of *patternNCC* and *patternNCS*.

Figure 5.12a that moving windows of different width from left to right (increasing the window starting column index) produces accuracy fluctuations that are very similar for different window width, this indicates that there is a core central part that is crucial for the correct classification through the *Search Pattern Routine*, the trend of the computation time instead results quite odd although ranging in a small window of values.



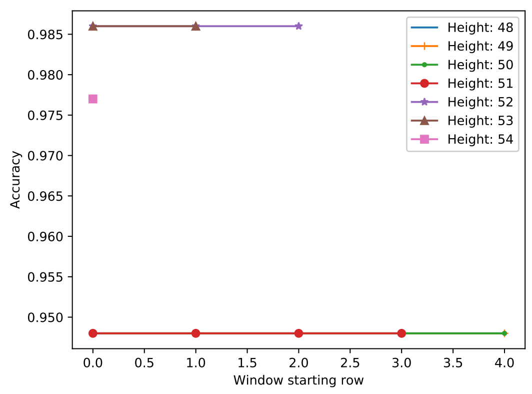
(a) Average percentage of correctly assigned captures.



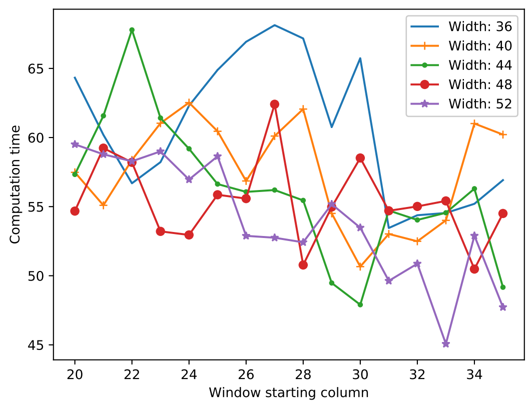
(b) Average time elapsed for classification.

Figure 5.12: Performance metrics for different *searchWindow* widths and horizontal positioning.

The accuracy trend for test *t13srcWdwHeight* instead, shows how crucial it is to keep the search window height as similar as possible to the viewport height: reducing the search window height value produces more misclassified captures regardless



(a) Average percentage of correctly assigned captures.



(b) Average time elapsed for classification.

Figure 5.13: Performance metrics for different *searchWindow* heights and vertical positioning.

the starting window row. This is not a surprise if we think to the nature of the *Search Pattern Routine* which exploits the informations deriving from layout modifications that move vertically the content of the page and thus requires as much informations as possible along the y axis.

5.2 Core Sections Detection parameter tuning

The core section detection algorithm requires less configuration parameters with respect to the layout detection system thus we were able to perform more extensive tests and evaluate a greater percentage of all possible configurations. The decision about the block side followed similar considerations with respect to what we said for the corresponding parameter of the layout detection algorithm. We decided to sacrifice a bit of computation time to obtain a more fine grained *changesMap* describing core sections in a more punctual way, we set the block side to 10 px. The most important parameters of the algorithm are *NN0Th* and *PDETh* on which we rely in order to take a final decision regarding each block. In *sdt1nn0pde* they were extensively tested and we can observe the outcome of these tests in Figure 5.14. We can notice that the more stable combination was the one with *PDETh* equal to 0.6 as it achieved the 100% of accuracy for a wide range of *NN0Th* values, we selected the central value of this range as the optimal parameter value.

In terms of computation time the section detection algorithm is deterministic, the computation cost decreases as we decrease the amount of examined captures. For this reason we decided to evaluate the performances in terms of accuracy with respect to the number of captures analysed in order to know if there was margin to limit the amount of captures evaluated keeping high levels of accuracy. The results

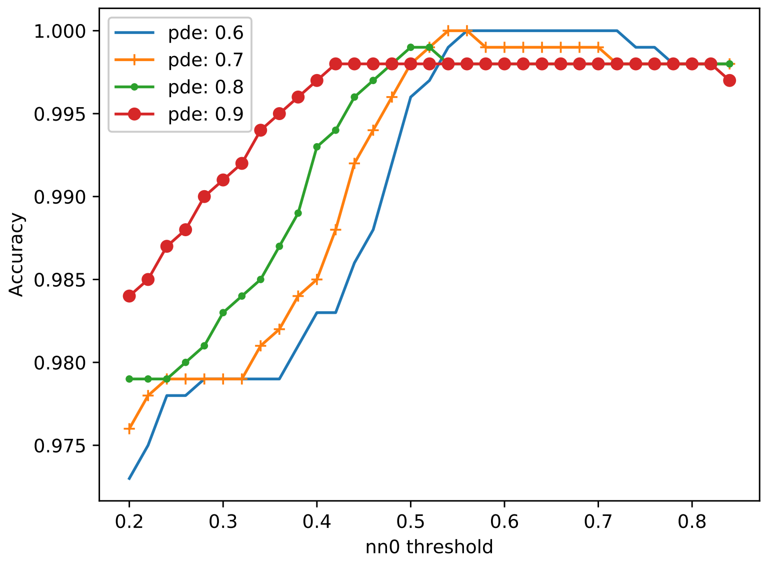


Figure 5.14: Performance metrics for different combinations of *NN0Th* and *PDETh*.

of test *sdt1captureLimit* presented in Figure 5.15 showed that, despite of course this kind of tests is influenced by the synthetic nature of the dataset, a set of 30 or more captures is enough to achieve the maximum possible accuracy.

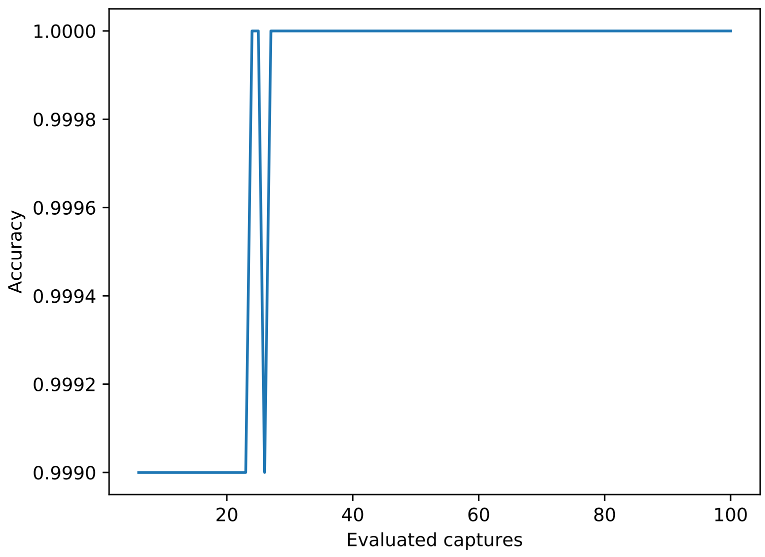
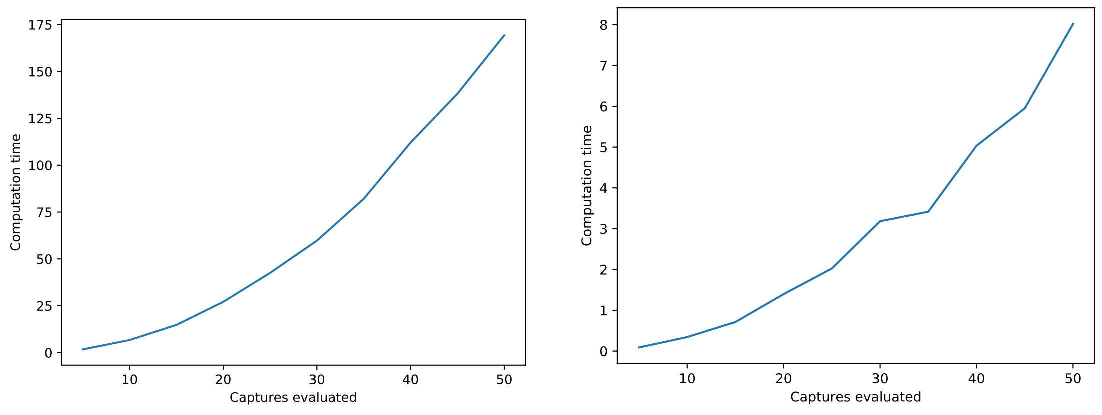


Figure 5.15: Performance metrics for different values of the *limitCaptures* parameter.

Finally in test *sdt2captureLimitTime* we examined the variations in terms of computation time (which as said is deterministic) when varying the amount of captures evaluated. The test has provided differentiated metrics for the capture analysis phase in which comparisons among captures are performed and of the final decision phase in which we take a decision regarding each block based on the analysis results and the provided thresholds and we produce the *changeMap*. The results are presented in Figure 5.16.



(a) Average time for capture analysis. (b) Average time for *changesMap* definition.

Figure 5.16: Computation cost for different phases of the core sections detection algorithm when varyin the amount of captures evaluated.

5.3 Final results summary

In this section we present the optimal configurations defined through over 150 hours of testing on the hardware described in section 4.6 for the layout detection and core sections detection algorithms. Along with the optimal configurations the related performance metrics for single thread runs over the same hardware configuration are presented.

Parameter name	**Tested in**	**Optimal Value**
blockWidth	t01blockSize	20 px
blockHeight	t01blockSize	20 px
NN0Th	t02nn0Th	0.133
highLvlMinDist	t03hlMinDist	0.15
highLvlMaxDist	t04hlMaxDist	0.83

coreWindowXStart	t05crWdwWidth	23
coreWindowXSpan	t05crWdwWidth	44
coreWindowYStart	t06crWdwHeight	3
coreWindowYSpan	t06crWdwHeight	46
coreLvlMinDist	t07clMinDist	0.19
coreLvlMaxDist	t08clMaxDist	0.55
patternNCC	t09ptrnConsec, t11ptrnConsShape	7
patternNCD	t10ptrnShape	2
patternNCS	t10ptrnShape, t11ptrnConsShape	10
srchWindowXStart	t12srcWdwWidth	30
srchWindowXSpan	t12srcWdwWidth	44
srchWindowYStart	t13srcWdwHeight	0
srchWindowYSpan	t13srcWdwHeight	53

Table 5.1: Best performing configuration for layout section detection algorithm.

The optimal configuration for the layout detection algorithm is summarized in Table 5.1. With this configuration, the test suite detailed in Table 4.3 and the dataset described in Table 4.1 the layout detection algorithm achieved the results in Table 5.2.

Metric	**Average value**	**Best result**	**Worst result**
Correctly Classified Captures	98.6%	100%	91.9%
Correctly Classified Layouts	95.1%	100%	71.4%
Elapsed Time	40.6 s	10.4 s	133.0 s
Perfect tests cases [1]	8/10	-	-

Table 5.2: Layout section detection algorithm performances for the optimal configuration.

[1]Presenting 100% of captures correctly classified and 100% of layouts detected.

For the core section detection algorithm we identified the optimal parameters summarized in Table 5.3. The *blockHeight* and *blockWidth* have been set to 10 px without extensive tests according to the desired granularity of the analysis. The presented configuration achieved the 100% of accuracy on the dataset detailed in Table 4.2. According to the results of *sdt2captureLimitTime* showed in Figure 5.16, the computation time for a set of 30 captures which has been identified in *sdt1captureLimit* as the optimal value for the capture set size, is of 59.74 seconds for captures analysis and 3.42 seconds for taking the final decision over each block and produce the output *changesMap*.

Parameter name	**Tested in**	**Optimal Value**
blockWidth	- [2]	10 px
blockHeight	-	10 px
NN0Th	sdt1nn0pde	0.65
PDETh	sdt1nn0pde	0.6
limitCaptures	sdt1captureLimit	30

Table 5.3: Best performing configuration for core sections detection algorithm.

[2]Provided apriori.

Chapter 6

Conclusions and Future Work

In the following we present some final considerations about this research work: first we discuss the tackled problem, its complexity and the progresses made towards its solution, then we further discuss the proposed solution, its advantages and possible limitations and the achieved performances. Finally we present possible directions of future works on this subject.

6.1 Considerations on the tackled problem

The aim of this research work was that of creating a tool able to identify with a general and content independent approach eventual "breaks" in web pages rendering when tracker-blocking tools are used. To the best of our knowledge, this specific task had never been addressed by any previous work; it has revealed several critical points: the complexity of deciding what does it mean for a web page top be broken, the variety of layouts showing different contents that the same web page can render and of course the difficult in finding a general purpose solution able to deal with the variety and mutability of the modern Internet.

The problem has not been fully solved but we were able to identify an effective approach to tackle this new genre of tasks and used this approach to address sub tasks of the main problem. We defined a visual approach based on the observation of the non broken version of the page in order to evaluate by comparison eventual rendering problems on that page. In particular we found a way to automatically collect captures of non broken web pages and visually identify their core contents, this enables understanding which elements must necessarily be properly rendered in order to characterize a page as not broken.

6.2 Our solution and possible improvements

Solving this sub problem required a system able to automatically collect captures with specific standardized characteristics of each web page, two algorithms able to group those captures according to the specific page layout they render and to visually identify the core content sections of the page by comparing captures rendering the same page layout. The advantage of the developed solution consists in being completely independent from the inner structure of the page, all the analysis is focused on the visual informations rendered when we visit the page which have always the same format and thus are easier to process in an organic way. This approach revealed to be fast and accurate in the vast majority of cases.

The algorithm for the detection and classification of captures layouts has achieved outstanding performances being able to properly identify the layouts rendered in 100 captures of the same web page with an average accuracy of 98.6% and an average computation time of 40.6 seconds on the limited hardware described in section 4.6. On the same hardware configuration, the algorithm for the core sections detection has reached the 100% of accuracy on all test cases with a computation time of 63.12 seconds for sets of 30 captures.

Our system has thus reached great results in terms of accuracy and time performances but there is still margin for further improvements and refining. First of all both the algorithms will require further and more extensive test: due to the limited amount of time available for this thesis work it was not possible to build large datasets to test the algorithms on a realistic sample of the variety of websites that it is possible to encounter in the modern Internet. It is crucial to extend the dataset described in Table 4.1 and provide for each test case and each layout the ground truth related to the core sections to be used in core sections detection tests, in fact the synthetic dataset exploited for the core sections detection algorithm testing and described in Table 4.2 can be considered only an approximation of real world cases.

The algorithmic part of the developed system has margin for improvements too, the nature of the captures comparison operation on which both algorithms are based suggests that performances could benefit, in terms of computation time, from a larger exploitation of parallelization in many subtasks.

6.3 Future work

The way paved by our visual approach, suggests which are the next steps that could be addressed by future works on this subject. In particular the visual map of the core sections of a web page produced as final output by the developed system can be used to locate and isolate within a page specific sections containing the most relevant page informations. To perform assertions about the health state of a page visited using tracker-blocking tools, future works could focus on locating this particular pieces of informations inside the evaluated pages by using image pattern recognition techniques.

To enable this kind of analysis, future studies will also also have to focus on identifying which visual transformation the core sections can undergo when tracker-blocking tools are activated.

Finally a system able to run autonomously the whole pipe of required operations will have to be designed, in such a way that it could respect all the requirements in terms of computation capabilities needed to operate on large sets of constantly changing and evolving web pages.

Appendix A

Tables of parameters

Here we present two section summarizing the various parameters of the developed algorithms, in order to ease the association of their name to their meaning and their role.

A.1 Layout Detection Parameters

In Table A.1 we summarize the parameters exploited by the layout detection routines.

Parameter name	**Meaning and role**
blockWidth	Width of the basic blocks in which we subdivide the captures to be analysed.
blockHeight	Height of the basic blocks in which we subdivide the captures to be analysed.
NN0Th	Number of Non Zero valued pixels: given a block derived from the subtraction of corresponding blocks of two capture, if the percentage of non zero valued pixels in it is higher than this threshold, we state that the block is a dynamic one this meaning that the two captures differ in that specific sub area.
highLvlMinDist	Maximum percentage of blocks differing among two captures in order to be assigned to the same layout for *High Level Similarity.*
highLvlMaxDist	Maximum percentage of blocks differing among two captures before declaring their layouts incompatible for *High Level non Similarity.*

coreLvlMinDist	Maximum percentage of blocks differing among two captures' core areas, as defined by the *coreWindow*, in order to be assigned to the same layout for *Core Level Similarity.*.
coreLvlMaxDist	Maximum percentage of blocks differing among two captures' core areas, as defined by the *coreWindow*, before declaring their layouts incompatible for *Core Level non Similarity.*
coreWindow	Parameter defining the size and shape of the window used to evaluate *Core Level Similarity.* It includes four sub parameters: **coreWindowXStart**, **coreWindowYStart** define the position of the first block included in the window; **coreWindowXSpan**, **coreWindowYSpan** define window width and height.
patternNCD	Number of Consecutive Dynamic blocks, parameter of the *Same Layout Pattern* presented in listing 3.3.3.
patternNCS	Number of Consecutive Static blocks, parameter of the *Same Layout Pattern* presented in listing 3.3.3.
patternNCC	Number of Consecutive Columns, parameter of the *Same Layout Pattern* presented in listing 3.3.3.
patternSrchWindow	Parameter defining the size and shape of the window in which we search the *Same Layout Pattern.* It includes four sub parameters: **srchWindowXStart**, **srchWindowYStart** define the position of the first block included in the window; **srchWindowXSpan**, **srchWindowYSpan** define window width and height.

Table A.1: Configuration parameters of layout detection algorithm.

A.2 Core Sections Detection Parameters

In Table A.2 we summarize the parameters exploited by the section detection algorithm.

Parameter name	**Meaning and role**
blockWidth	Width of the basic blocks in which we subdivide the captures to be analysed.
blockHeight	Height of the basic blocks in which we subdivide the captures to be analysed.
NN0Th	Number of Non Zero valued pixels: given a block derived from the subtraction of corresponding blocks of two capture, if the percentage of non zero valued pixels in it is higher than this threshold, the block is evaluated as dynamic in the context of that captures comparison.
PDETh	Percentage of Dynamic Evaluations: given the results of a set of comparisons each one providing a partial evaluation about the block nature, if a block is evaluated to be dynamic in a percentage of comparisons greater than **pdeTh**, than it is definitively evaluated as dynamic.
limitCaptures	Parameter limiting the number of captures involved in comparisons operations.

Table A.2: Configuration parameters of section detection algorithm.

Bibliography

[1] Y. Yuan, F. Wang, J. Li, R. Qin "A survey on real time bidding advertising", Proceedings of 2014 IEEE International Conference on Service Operations and Logistics, and Informatics, 2014, DOI 10.1109/SOLI.2014.6960761

[2] S. C. Boerman, S. Kruikemeier, F. J. Zuiderveen Borgesius, "Online Behavioural Advertising: A Literature Review and Research Agenda", in Journal of Advertising, 2017, DOI 10.1080/00913367.2017.1339368

[3] A. Barth, U.C. Berkeley, "HTTP State Management Mechanism", Internet Engineering Task Force, in RFC-6265, April 2010,

[4] A. Soltani, S. Canty, Q. Mayo, L. Thomas, C. J. Hoofnagle "Flash Cookies and Privacy", in Summer Undergraduate Program in Engineering Research at Berkeley (SUPERB), Berkley (USA), 2009

[5] S. Kamkar, Evercookie, `https://samy.pl/evercookie/`

[6] P. Eckersley, "How Unique Is Your Browser?", in Proceedings of the 10th Privacy Enhancing Technologies Symposium (PETS), 2010

[7] K. Mowery, D. Bogenreif, S. Yilek, H. Shacham, "Fingerprinting information in JavaScript implementations", in Proceedings of W2SP 2011, IEEE Computer Society, May 2011

[8] K. Mowery, H. Shacham, "Pixel perfect: Fingerprinting canvas in HTML5", in Proceedings of W2SP 2012, M. Fredrikson, IEEE Computer Society, May 2012

[9] J. R. Mayer, J. C. Mitchell, "Third-Party Web Tracking: Policy and Technology", in 2012 IEEE Symposium on Security and Privacy, May 2012, DOI 10.1109/SP.2012.47

[10] "Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation)", in Official Journal of the European Union, May 2016, `http://eur-lex.europa.eu/legal-content/EN/TXT/?uri=OJ:L:2016:119:TOC`

[11] "Tracking Preference Expression (DNT)", in W3C Working Group Note, January 2019, `https://www.w3.org/TR/tracking-dnt/`

[12] R. Balebako, P. Leon, R. Shay, B. Ur, Y. Wang L. Cranor, "Measuring the

effectiveness of privacy tools for limiting behavioral advertising" in W2SP, 2012.

[13] J.Mayer, "Fourthparty web measurement platform", 2015, `http://fourthparty.info/`

[14] D. Reisman, S. Englehardt, C. Eubank, P. Zimmerman, A. Narayanan, "Cookies that give you away: Evaluating the surveillance implications of web tracking", in WWW, 2014

[15] G. Merzdovnik et al., "Block Me If You Can: A Large-Scale Study of Tracker-Blocking Tools", in 2017 IEEE European Symposium on Security and Privacy, Paris, 2017, DOI 10.1109/EuroSP.2017.26

[16] S. Traverso, M. Trevisan, l. Giannantoni, M. Mellia, H. Metwalley, "Benchmark and comparison of tracker-blockers: Should you trust them?", in 2017 Network Traffic Measurement and Analysis Conference (TMA), June 2017, DOI 10.23919/TMA.2017.8002898

[17] Alberto H. F. Laender, Berthier A. Ribeiro-Neto, Altigran S. da Silva, Juliana S. Teixeira, "A Brief Survey of Web Data Extraction Tools", in SIGMOD Rec. 31, June 2002, DOI 10.1145/565117.565137

[18] G.O. Arocena, A.O, Mendelzon, "WebOQL: Restructuring documents, databases and Webs", in Proc. ICDE 98., 1998, DOI 10.1109/ICDE.1998.655754

[19] A. Sahuguet, F. Azavant, "Building intelligent web applications using lightweight wrappers", in Data Knowledge Engineering 36, March 2001, DOI 10.1016/S0169-023X(00)00051-3

[20] Soderland, Stephen. " Learning Information Extraction Rules for Semi-Structured and Free Text", in Machine Learning 34, 1999, DOI 10.1023/A:1007562322031

[21] I. Muslea, S. Minton, C. A. Knoblock, "Hierarchical Wrapper Induction for Semistructured Information Sources", in Autonomous Agents and Multi-Agent Systems, March 2001, DOI 10.1023/A:1010022931168

[22] A. Laender, B. Ribeiro-neto, S. Altigran, "DEByE - Data extraction by example", in Data Knowledge Engineering 40, 2002, DOI 10.1016/S0169-023X(01)00047-7

[23] "Ontologies", in W3C Standards, Semantic Web, `https://www.w3.org/standards/semanticweb/ontology`

[24] M. Fernandez, I. Cantador, V. Lopez, D. Vallet, P. Castells, E. Motta, "Semantically enhanced Information Retrieval: an ontology-based approach", in Web Semantics: Science, Services and Agents on the World Wide Web 9, 2011, `http://www.websemanticsjournal.org/index.php/ps/article/view/242`

[25] J. Alarte, D. Insa, J. Silva, S. Tamarit, "Web Template Extraction Based on Hyperlink Analysis", in EPTCS 173, 2015, DOI 10.4204/EPTCS.173.2

[26] Z. Bar-Yossef, S. Rajagopalan, "Template Detection via Data Mining and Its

Applications", in 11th International Conference on World Wide Web, Honolulu, Hawaii, USA 2002, DOI 10.1145/511446.511522

[27] F. Sun, D. Song, L. Liao, "DOM Based Content Extraction via Text Density", in 34th International ACM SIGIR Conference on Research and Development in Information Retrieval, Beijing, China, 2011, DOI 10.1145/2009916.2009952

[28] L. Yi, B. Liu, X. Li, "Eliminating Noisy Information in Web Pages for Data Mining", in 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Washington, D.C., USA, August 2003, DOI 10.1145/956750.956785

[29] L. J. Martinez-Rodriguez, A. Hogan, I. Lopez-Arevalo, Ivan "Information extraction meets the Semantic Web: A survey", in Semantic Web journal, October 2018, DOI 10.3233/SW-180333

[30] "The Semantic Web Made Easy", in Semantic Web journal, October 2018, DOI 10.3233/SW-180333

[31] H. Alani, D. E. Millard, M. J. Weal, W. Hall, P. H. Lewis, N. R. Shadbolt, "Automatic ontology-based knowledge extraction from Web documents", in IEEE Intelligent Systems 18, January 2003, DOI 10.1109/MIS.2003.1179189

[32] M. Bernard, "Criteria for optimal web design (designing for usability)", 2003

[33] C. Deng, Y. Shipeng, W. Ji-Rong, M. Wei-Ying, "VIPS: a Vision-based Page Segmentation Algorithm", 2003

[34] M. Ester, H. P. Kriegel, J. Sander, X. Xu, "A Density-based Algorithm for Discovering Clusters in Large Spatial Databases with Noise", in Proceedings of the Second International Conference on Knowledge Discovery and Data Mining, Portland, Oregon, 1996 `http://dl.acm.org/citation.cfm?id=3001460.3001507`