

ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS

# Crawling Greek business websites

---

Master in Data Science



**Author:** Kapsalis Panagiotis

**Supervisor:** Vassalos Vasileios

Athens, October 2017

## **Abstract**

Internet is a source of live data that is constantly updating with data of almost any field we can imagine. Having tools that can automatically detect these updates and can select that information that we are interested in are becoming of utmost importance nowadays. So that is the reason we implement crawlers for Greek businesses, in order to extract elements from their websites. Crawling and text engineering methods are applied in order to extract business data. The purpose of this diploma thesis is to investigate how feasible is to extract information about companies from their websites.

## Contents

Chapter 1: Introduction .....	7
1.1 Web scraping.....	7
1.2 Web scraping techniques .....	8
Text pattern matching.....	8
HTTP programming.....	8
HTML parsing.....	8
DOM parsing.....	8
Vertical aggregation .....	8
Semantic annotation recognizing.....	9
Software tools .....	9
1.3 Focused Crawling.....	9
1.4 Aim of the thesis.....	10
1.5 Ethical Considerations .....	10
Chapter 2: Tools and Technologies.....	11
2.1 Scientific approach .....	11
2.2 Project's Approach .....	12
2.3 Python Libraries.....	12
Requests Library in Python.....	12
Beautiful Soup .....	13
Python Regular Expressions .....	15
Selenium WebDriver.....	16
Geopy .....	18
Urllib.parse module .....	19
2.4 Online Tools.....	20
Internet Archive.....	20
Google Places API .....	21
Google Maps Geocoding API .....	22
Statsshow.com .....	23
Google Page Speed Insights API .....	24
Chapter 3: Implementation and Results.....	26
3.1 First Crawler .....	26
3.2 Second crawler .....	28
3.3 Third crawler.....	33

3.4 Fourth crawler .....	35
3.5 Fifth crawler.....	37
3.6 Problems.....	40
3.7 Results .....	43
References.....	48

## Table of Figures

Figure 1 Install Requests Library.....	13
Figure 2 Using Requests Library .....	13
Figure 3 Install Beautiful Soup library .....	14
Figure 4 Example using Beautiful Soup .....	14
Figure 5 Install Selenium .....	16
Figure 6 Fetch webpage using Selenium .....	17
Figure 7 XPath Tree .....	17
Figure 8 XPath Syntax.....	18
Figure 9 Internet Archive.....	20
Figure 10 Internet Archive example .....	20
Figure 11 Statsshow.com .....	24
Figure 12 the Use of statsshow.com .....	24
Figure 13 Google Insights API.....	25
Figure 14 First Crawler demo .....	26
Figure 15 First crawler evaluation .....	27
Figure 16 Email crawler example .....	28
Figure 17 Find certifications example .....	28
Figure 18 finder function demo.....	30
Figure 19 find_bname function demo 1.....	30
Figure 20 find_bname function demo 2.....	30
Figure 21 find_bname function demo 3.....	31
Figure 22 HTML document title.....	31
Figure 23 find_bname function demo 3.....	31
Figure 24 country_data demo 1 .....	31
Figure 25 country_data demo 2 .....	32
Figure 26 Second Crawler evaluation.....	33
Figure 27 Website Development stats .....	33
Figure 28 wayback machine function example .....	34
Figure 29 advanced stats function example.....	34
Figure 30 Second crawler results .....	35
Figure 31 tk_search function.....	36
Figure 32 tk_search demo .....	36
Figure 33 Find place function .....	36
Figure 34 Find Place demo .....	36
Figure 35 Fourth crawler demo.....	36
Figure 36 Fourth Crawler results.....	37
Figure 37 etke_exp_imp function demo 1 .....	38
Figure 38 etke_exp_imp function demo 2 .....	38
Figure 39 etke_exp_imp function demo 3 .....	38
Figure 40 rep_sup_fac function demo 1 .....	39
Figure 41 rep_sup_fac function demo 2 .....	39
Figure 42 rep_sup_fac function demo 3 .....	39
Figure 43 find_awards function demo .....	39
Figure 44 Fifth crawler results.....	40

Figure 45 URL form.....	40
Figure 46 Connection Error .....	41
Figure 47 404 error.....	41
Figure 48 adobe flash applications.....	41
Figure 49 403 Error.....	42
Figure 50 User Agents .....	42
Figure 51 Urls and Status codes .....	43
Figure 52 Correct extraction for each field .....	44

# Chapter 1: Introduction

Data Science is changing the world with its capabilities to identify trends, predict the future and derive deep insights like never before from large data sets. It is understood that data is the fuel for any data science related project. Since web is becoming the biggest repository of data has ever been, it makes sense to consider web scraping for fueling data science use cases. This thesis is about Web crawling, the process used by Web search engines to download pages from the Web. This opening chapter introduces web scraping.

## 1.1 Web scraping

Web scraping (web harvesting or web data extraction) is data scraping used for extracting data from websites. Web scraping software may access the World Wide Web directly using the HTTP, or through a web browser. While web scraping can be done manually by a software user, the term typically refers to automated processes implemented using a bot or web crawler. It is a form of copying, in which specific data is gathered and copied from the web, typically into a central local database or spreadsheet, for later retrieval or analysis.

Scraping a web page involves fetching it and extracting from it. Web crawling is a main component of web scraping, to fetch pages for later processing. Once fetched, then extraction can take place. The content of a page may be parsed, searched, reformatted, its data copied into a spreadsheet, and so on. Scrapers typically take something out of a page, to make use of it for another purpose somewhere else. An example would be to find and copy names, phone numbers, or companies and their URLs.

Web pages are built using text-based mark-up languages (HTML and XHTML), and frequently contain a wealth of useful data in text form. However, most web pages are designed for human end-users and not for ease of automated use. Because of this, tool kits that scrape web content were created. A web scraper is an API to extract data from a web site. Companies like Amazon AWS and Google provide web scraping tools, services and public data available free of cost to end users. Newer forms of web scraping involve listening to data feeds from web servers. For example, JSON is commonly used as a transport storage mechanism between the client and the web server.

There are methods that some websites use to prevent web scrapping, such as detecting and disallowing bots from crawling (viewing) their pages. In response, there are web scraping systems that rely on using techniques in DOM parsing, computer vision and

natural language processing to simulate human browsing to enable gathering web page content for offline parsing.

## 1.2 Web scraping techniques

Current web scraping solutions range from the ad-hoc, requiring human effort, to fully automated systems that are able to convert entire websites into structured information, with limitations.

### Text pattern matching

A simple yet powerful approach to extract information from web pages can be based on the UNIX grep command or regular expression-matching facilities of programming languages.

### HTTP programming

Static and dynamic pages can be retrieved by posting HTTP requests to the remote web server using socket programming.

### HTML parsing

Many websites have large collections of pages generated dynamically from an underlying structured source like a database. Data of the same category are typically encoded into similar pages by a common script or template. In data mining, a program that detects such templates in a particular information source, extracts its content and translates it into a relational form, is called wrapper. Wrapper generation algorithms assume that input pages of a wrapper induction system conform to a common template and that they can be easily identified in terms of a URL common scheme. Moreover, some semi structured query languages, such as XQuery and the HTQL, can be used to parse HTML pages and to retrieve and transform page content.

### DOM parsing

By embedding a full fledged web browser, such as the Internet Explorer or the Mozilla browser control, programs can retrieve the dynamic content generated by client side scripts. These browser controls also parse web pages into a DOM tree, based on which programs can retrieve parts of the pages.

### Vertical aggregation

There are several companies that have developed vertical harvesting platforms. These platforms create and monitor a multitude of bots for specific verticals with no man in the loop (no direct human involvement), and no work related to a specific target site. The preparation involves establishing the knowledge base for the entire vertical and then the platform creates the bots automatically. The platform's robustness is measured by the quality of the information it retrieves (usually number of fields) and its scalability (how quick it can scale up to hundreds or thousands of sites.) This scalability is mostly used to target the Long Tail of sites that common aggregators find complicated or too labor – intensive to harvest content from.



### Semantic annotation recognizing

The pages being scraped may embrace metadata or semantic markups and notations, which can be used to locate specific data snippets. If the annotations are embedded in the pages, as Microformat does, this technique can be viewed as a special case of DOM parsing. In another case, the annotations, organized into a semantic layer, are stored and managed separately from the web pages, so the scrapers can retrieve data schema and instructions from this layer before scraping the pages.

### Software tools

There are many software tools available that can be used to customize web-scraping solutions. This software may attempt to automatically recognize the data structure of a page or provide a recording interface that removes the necessity to manually write web-scraping code, or some scripting functions that can be used to extract and transform content, and database interfaces that can store the scraped data in local databases. Some web scraping software can be also used to extract from an API directly. Below we will list some of software tools that have been considered quite popular

- Scrapy: This is a free and open source web crawling framework for python. It can be used for scraping or for general crawling. Within this project, it would have the disadvantage of using python instead of Java, which is the language used for the whole LnuDSC project. Also it is not clear if it supports AJAX or dynamic websites in order to extract its information. While testing this application, it offers plenty of possibilities and tools. Through an application implemented using this framework it was only possible to extract the desired information available in that page. However this framework, does not provide any method to interact with the website.
- Import.io: This tool, is not free. In opposition to the previous tool (Scrapy) import.io provides its own API and is not instead for pure developers, but for being used as an standalone application that already extracts the desired data. Is this one the main reason why it is not appropriate tool for this project, including also that the possibilities to extract data from dynamic or AJAX websites was not found.

### 1.3 Focused Crawling

The importance of a page for a scraper can also be expressed as a function of the similarity of a page to a given query. Web crawlers (scrapers) that attempt to download pages that are similar to each other are called focused or topical crawlers. The main problem in focused crawling is that the context of a web scraper, we would like to be able to predict the similarity of the text of a given page to the query before actually downloading the page. A possible predictor is the anchor text of links, this was the approach taken by Pinkerton in a crawler developed in the early days of the Web. The performance of a focused scraper depends mostly on the richness of links in

the specific topic being searched, and a focused scraper usually relies on a general Web search engine for providing starting points.

## 1.4 Aim of the thesis

The goal of the thesis is to develop methodologies and technologies crawling data from the business greek domain, and more specifically the construction of a focused crawler (scraper), which extracts data from sites belonging on new enterprises, hotels etc. This needs to identify first good seed sites, which can include for example information about business activities such as in which countries an enterprise exports its goods, email or phone contacts, geographical coordinates of business location, qualities certificates and many others. To export the data we need from the unstructured information, we need to automatically identify common patterns on websites, this can be achieved by using regular expression, online tools such as Wayback Machine to identify website's last modified date or third party APIs, such as geocoding APIs to identify business location etc. Chapter 2 refers to the tools (Python libraries, APIs, online tools) that have been used for the development of the crawler.

## 1.5 Ethical Considerations

Crawling and extracting information from the Web is quite a common practice that is being done by many companies all around the world. It has a lot of obvious benefits that have been already discussed, but they do not come with some ethical and even legal drawbacks. Most of the websites in the Internet are public, which means that anyone is allowed to enter and consume its content, but also many of them keep some copyrights that are not always respected while crawling or extracting data with automatic robots as the one that is being implemented through this thesis.

This is a topic that depends a lot on the kind of content that different websites might provide and how the information extracted from those will be used. For example, there some specific tools called aggregators that provide content extracted from other different websites. Those aggregators can be really useful, but using them means not using the official websites that provide this information. Those original websites might get benefits from the number of visits their website has. Taking these facts into account, it can conform an ethical, in some situations, even a legal problem.

A well known issue related to this is the new aggregators that have been forbidden in some countries. For example, recently Google News, a very well known new aggregator, decided to close and not provide this tool to some countries where these kind of tools, due to the problem mentioned before, were asked to pay a tax.

## Chapter 2: Tools and Technologies

In this chapter we will describe the scientific approaches that we followed and introduce tools and technologies that have been used.

### 2.1 Scientific approach

The selected scientific approach for this thesis will involve mostly quantitative methods. As mentioned in the introduction (Chapter 1) we will try to develop a working scraper, who will gather specific fields from websites. More specifically the scraper must extract from each website he visits, email and phone contacts which are mentioned on the website, the company's location (geographical coordinates, zip code, street address), countries on which the company operates, quality assurance, if the company has been awarded etc.

In a first step, the analysis of the different websites is needed and we need to have a deeper understanding of them. In order to implement our crawler, an iterative methodology will be followed. This means that in a first step we will try to develop a working robot to crawl a specific website. Afterwards we will check possible errors and fix them, introducing new code from the results obtained previously. The next step is to replicate this process for different sites, in order to generalize its functionality. To achieve better results, we will study how websites work. Inspecting their available source code (HTML, JavaScript), we will try to understand which processes the websites follow to provide data, in order to find the pattern to extract the required information.

In order to have valid and reliable data, a whole process of verification and study of the crawler should be made. First of all the crawler should be tested repeatedly under the same conditions, considering same time frames, same internet connection and same working load on the computer that runs the program. Once done this, the obtained data should be compared both between them and also with the raw data of the website. This would verify the correctness of the extracted data by the implemented crawler. By this approach, we would be able to state the reliability and validity of this tool, as well as its drawbacks. However, some of these conditions are really difficult to repeat as we do not have the resources to control them. For example, we are not able to have complete control on the working load of the computer in two different runs of the crawler, as many system processes might change without us noticing. Also the internet connection, or more precisely the whole connection and load of the server in order to have control of the response times, is something impossible to completely control.

In order to achieve performance, it is better the element extraction not to be done from only one crawler. As a result we must distribute the workload in more than one program. So it would be as follows:

- First crawler: It checks if a website has Multilanguage option, extracts the social networks which are included in this website and its own URLs, it

checks if the website has blog, newsletter, search option, e-shop and mobile application

- Second crawler: It checks if representants, exports, imports, customer support, corporate social responsibility have been referred in the website
- Third crawler: Extracts the quality certificates referred to the website and business email and phone contacts.
- Fourth crawler: It extracts the countries in which the business operates.
- Fifth crawler: It extracts business location, more specifically company's geographical coordinates, street address and zipcode
- Moreover, to take website's last modified date and some information about website's traffic, have been used online tools such as Way back Machine, statsshow.com and siteworthtraffic.com.

## 2.2 Project's Approach

In order to verify application's correct functionality, we choose to make this program to run for 5000 websites, which belong to business Greek domain, retrieving their elements. Doing that it is possible to verify its correct functionality as well as to find possible errors that might occur.

Once obtained the data, it is needed to take a look both to the data and website's raw data, to verify application's correct functionality. In order to do so, as the amount of data is too big to validate it all, we will just take an overview, focusing on where some problems might be expected, but also understanding that if the scraper works well for an amount of websites, it can be considered that there should not be any further problems. This is because the websites maintain a common structure along their many pages, so we can expect the application to work the same way with one page and with the rest of them. But the problem presents to find the pattern among the websites, in order to navigate on pages that belong on many different sites. For the final step, in order to validate our work, we will take a random sample of websites and check the validity of application results with their content.

## 2.3 Python Libraries

For developing this scraper the following python libraries and modules have been used:

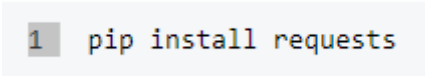
- Requests 2.18.4
- BeautifulSoup 4.6.0
- Regex module
- Geopy 1.11.0
- Selenium 3.5.0

### Requests Library in Python

Requests is an Apache2 Licensed HTTP library, written in Python. It is designed to be used by humans to interact with the language. This means you don't have to manually add query strings to URLs, or form-encode your POST data. Requests will allow you

to send HTTP/1.1 requests using Python. With it, you can add content like headers, form data, multipart files, and parameters via simple Python libraries. It also allows you to access the response data of Python in the same way. In programming, a library is a collection or pre – figured selection of routines, functions and operations that a program can use. These elements are often referred to as modules, and stored in object format. Libraries are important, because you load a module and take advantage of everything it offers without explicitly linking to every program that relies on them. They are truly standalone, so you can build your own programs with them and yet they remain separate from other programs.

They are a few ways to install the Requests library, via pip or easy\_install, or tarball. Bellow the image demonstrates Requests installation using pip.



```
1 pip install requests
```

Figure 1 Install Requests Library

Requests will automatically decode any content pulled from a server, but most of Unicode character sets are seamlessly decoded anyway. When you make a request to a server, the Requests library makes an educated guess about the encoding for the response, and it does this based on the HTTP headers. The encoding that is guessed will be used when you access the `r.text` file. Also we must mention that there are a number of exceptions and error codes you need to be familiar with, when using the Requests library in Python. If there is a network problem like a DNS failure, or refused connection the Requests library will raise a `ConnectionError` exception. With invalid HTTP responses, Requests will also raise an `HTTPError` exception, but these are rare. If a requests times out, a `TimeoutException` will be raised and if and when a request exceeds the preconfigured number of maximum redirections, then a `TooManyRedirects` will be raised. Bellow the images shows how to make an HTTP request, and how to use some Requests modules.



```
01 import requests
02 req = requests.get('http://www.tutsplus.com/')
03
04 req.encoding      # returns 'utf-8'
05 req.status_code   # returns 200
06 req.elapsed       # returns datetime.timedelta(0, 1, 666890)
07 req.url           # returns 'https://tutsplus.com/'
08
09 req.history
10 # returns [<Response [301]>, <Response [301]>]
11
12 req.headers['Content-Type']
13 # returns 'text/html; charset=utf-8'
```

Figure 2 Using Requests Library

## Beautiful Soup

Beautiful Soup is a Python library for getting data out of HTML, XML, and other markup languages. For example, there are some pages that display data we want, such as date or address information, but that do not provide any way of downloading the

data directly. BeautifulSoup helps you pull particular content from a webpage, remove the HTML markup, and save the information. It is a tool for web scraping that helps you clean up and parse the documents you have pulled down from the web.

Installing BeautifulSoup is easiest if you have pip or another Python installer already in place. Once you have pip installed, run the following command in the terminal to install the library:

```
pip install beautifulsoup4
```

Figure 3 Install BeautifulSoup library

To scrape a webpage using BeautifulSoup, the first thing to do is getting a copy of the HTML page want to scrape. In this project we combine Requests library with BeautifulSoup, to take their content and then to scrape them. One of the first things this library can help us, is locating content in website's HTML structure. BeautifulSoup allows to select content based upon tags. Bellow the image shows how to use the library among Requests library to fetch data:

```
from bs4 import BeautifulSoup

r = requests.get("https://en.wikipedia.org/wiki/HTML")

html = r.content
soup = BeautifulSoup(html, 'html.parser')
print(soup.h1)

<h1 class="firstHeading" id="firstHeading" lang="en">HTML</h1>
```

Figure 4 Example using BeautifulSoup

The BeautifulSoup class is full of web-browser-like heuristics for divining the intent of HTML authors. But XML doesn't have a fixed tag set, so those heuristics don't apply. So BeautifulSoup doesn't do XML very well. Use the BeautifulSoup class to parse XML documents. The most common short coming of BeautifulSoup is that it doesn't know about self-closing tags. HTML has a fixed set of self-closing tags, but with XML it depends on what the DTD says. You can tell BeautifulSoup that certain tags are self-closing by passing in their names as the self Closing Tags argument to the constructor:

The findAll() method traverses the tree, starting at the given point, and finds all the Tag and NavigableString objects that match the criteria you give. The signature for the findall() method is this:

**findAll(name=None, attrs={}, recursive=True, text=None, limit=None, \*\*kwargs)**

These arguments show up over and over again throughout the BeautifulSoup API. The most important arguments are name and the keyword arguments. The name argument restricts the set of tags by name. There are several ways to restrict

the name, and these too show up over and over again throughout the BeautifulSoup API. For example using `findAll()` function we have:

```
1. soup.findAll('b')
2. # [<b>one</b>, <b>two</b>]
3. You can also pass in a regular expression. This code finds all the tags whose
   names start with B:
4. import re
5. tagsStartingWithB = soup.findAll(re.compile('^b'))
6. #[tag.name for tag in tagsStartingWithB]
7. # ['body', 'b', 'b']
```

## Python Regular Expressions

A regular expression is a special sequence of characters that helps you match or find other strings or set of strings, using a specialized syntax held in a pattern. Regular expressions are widely used in UNIX world. The module `re` provides full support for Perl like expressions in Python. The `re` module raises the exception `re.error` if an error occurs while compiling or using a regular expression. We would cover two important functions, which would be used to handle URLs, among BeautifulSoup, from which our crawler will extract some elements.

There are various characters, which would have special meaning when they are used in regular expression. To avoid any confusion while dealing with regular expressions, we would use Raw Strings as `r'expression'`. The **`match()`** function attempts to match pattern to string with optional flags. The syntax for this function is the following:

```
re.match(pattern, string, flags=0)
```

The `re.match` function returns a match object on success, `None` on failure. The `search()` function searches for first occurrence of RE pattern within string with optional flags. The `re.search` function returns a match object on success, `none` on failure, here is the syntax:

```
re.search(pattern, string, flags=0)
```

Python offers two different primitive operations based on regular expressions: **`match`** checks for a match only at the beginning of the string, while `search` checks for a match anywhere in the string. Except for control characters, `(+ ? . * ^ $ ( ) [ ] { } | \)`, all characters match themselves, you can escape a control character by preceding it with backslash. Bellow, we analyze the use of each control characters:

- The character `^` matches the beginning of a line.
- The character `$` matches the end of a line.
- The characters `[` and `]` define character classes
- `[a-z]` defines the characters from a to z.



- `.` matches any character.
- `\d` matches any decimal digit; this is equivalent to the class `[0-9]`.
- `\D` matches any non-digit character; this is equivalent to the class `[^0-9]`.
- `\S` matches any non-whitespace character; this is equivalent to the class `[^\t\n\r\f\v]`.
- `\w` matches any alphanumeric character; this is equivalent to the class `[a-zA-Z0-9_]`.
- `\W` matches any non-alphanumeric character; this is equivalent to the class `[^a-zA-Z0-9_]`.
- `\` escapes a character.

More specifically, in this project we use `re` module with Beautiful Soup to control the url in which the crawler goes to extract elements. For example we want the application to crawl the business email and phone contact from website's hrefs which contains the string 'contact' or 'contact us' or 'about' etc.

## Selenium WebDriver

Selenium is a set of different software tools each with a different approach to supporting test automation. The entire suit of tools results in a rich set of testing functions specifically geared to the needs of testing of web applications of all types. These operations are highly flexible, allowing many options for locating UI elements and comparing expected test results against actual application behavior. One of Selenium's key features is the support for executing one's tests on multiple browser platforms. Selenium is composed of multiple software tools. Each has a specific role, but in this project we will use Selenium 2 or Selenium WebDriver.

The primary new feature in Selenium 2 is the integration of the WebDriver API. WebDriver is designed to provide a simpler, more concise programming interface in addition to addressing some limitations in the Selenium RC API. Selenium WebDriver was developed to better support dynamic web pages where elements of a page may chnge without the page itself being reloaded, for example embedded google maps on business website. WebDriver's goal is to supply a well designed object API that provides improved support for modern advanced web app testing problems.

Selenium WebDriver makes direct calls to the browser using each browser's native support for automation. These calls and the features they support depends on the browser on the browser we use. To add Selenium to Python environment, we must run the following command:

```
pip install selenium
```

Figure 5 Install Selenium



The first thing we want to do is navigate to a page, the normal way to do this is by calling `get()` function:

```
driver.get("http://www.google.com")
```

Figure 6 Fetch webpage using Selenium

Locating elements in WebDriver can be done on the WebDriver instance itself or on a WebElement. Each of the language binding exposes a “Find Element” and “Find Elements” method. The former returns a WebElement object matching the query, and throws an exception if such an element cannot be found. The latter returns a list of WebElements, possibly empty if no DOM elements match the query. The “Find” methods take a locator or query object called “By”. Selenium fetches DOM elements using the following strategies

- By ID
- By Class Name
- By Tag Name
- By Name
- By Link Text
- By Partial Link Text
- By CSS
- By XPath

XPath uses path expressions to select nodes or node – sets in an XML document. There are functions for string values, numeric values, Booleans, date and time comparison, node manipulation, sequence manipulation, and much more. In XPath, there are seven kinds of nodes: element, attribute, text, namespace, processing – instruction, comment and document nodes. XML documents are treated as trees of nodes. The topmost element of the tree is called the root element. Each element and attribute has one parent, element nodes may have zero, one or more children. In the following example, the book element is the parent of the title, author, year and price. Also the title, author and price elements are all children of the book element:

```
<book>
  <title>Harry Potter</title>
  <author>J K. Rowling</author>
  <year>2005</year>
  <price>29.99</price>
</book>
```

Figure 7 XPath Tree

Expression	Description
<i>nodename</i>	Selects all nodes with the name " <i>nodename</i> "
/	Selects from the root node
//	Selects nodes in the document from the current node that match the selection no matter where they are
.	Selects the current node
..	Selects the parent of the current node
@	Selects attributes

Figure 8 XPath Syntax

In this project to fetch UI elements we use the “By XPath” strategy, the following command finds all input elements.

## Geopy

Geopy is Python’s Geocoding Toolbox, more specifically is a Python 2 and 3 client for several popular geocoding web services. Geopy makes it easy for Python developers to locate the coordinates of addresses, cities, countries, and landmarks across the globe using third party geocoders and other data sources, it is also tested against CPython 2.7, CPython 3.2, CPython 3.4, PyPy and PyPy3.

Each geolocation service you might use, such as Google Maps, Bing Maps or Yahoo BOSS, has its own class in `geopy.geocoders` abstracting the service’s API. Geocoders each define at least a `geocode` method, for resolving a location from a string and may define a `reverse` method, which resolves a pair of coordinates to an address. Each Geocoder accepts any credentials or settings needed to interact with its services, an API key or locale, during its initialization. To geolocate a query to an address and coordinates:

```
>>> from geopy.geocoders import Nominatim
>>> geolocator = Nominatim()
>>> location = geolocator.geocode("175 5th Avenue NYC")
>>> print(location.address)
Flatiron Building, 175, 5th Avenue, Flatiron, New York, NYC,
New York, ...
>>> print((location.latitude, location.longitude))
(40.7410861, -73.9896297241625)
>>> print(location.raw)
{'place id': '9167009604', 'type': 'attraction', ...}
```

To find the address corresponding to a set of coordinates:

```
>>> from geopy.geocoders import Nominatim
>>> geolocator = Nominatim()
>>> location = geolocator.reverse("52.509669, 13.376294")
>>> print(location.address)
Potsdamer Platz, Mitte, Berlin, 10117, Deutschland, European
Union
>>> print((location.latitude, location.longitude))
(52.5094982, 13.3765983)
>>> print(location.raw)
{'place_id': '654513', 'osm_type': 'node', ...}
```

Locator's geolocate and reverse methods require the argument query, and also accept at least the argument "exactly\_one", which is True. Geocoders may have additional attributes, e.g., Bing accepts "user\_location", the effect of which is to bias results near that location, geolocate and reverse methods may return three types of values:

- When there are no results found, returns "None".
- When the method's "exactly\_one" argument is True and at least one result is found, returns a "geopy.location.Location" object, which can be iterated over as (address,(latitude, longitude)), or can be accessed as Location.address, Location.latitude, Location.longitude.
- When exactly\_one is False, and there is at least one result, returns list of geopy.location.Location object as [Location,...]

If a service is unavailable or otherwise returns a non – OK response, or doesn't receive a response in the allotted timeout, you will receive Timeout Geocoder Exception. In that case we can extend the timeout parameter, or to use another geocoding service, in this project when Timeout exception is raised, then google geolocation API is called.

## Urllib.parse module

To navigate on website links, we use urllib.parse library and more specifically urljoin and urlparse methods. Urllparse() method parse a url into six components, returning a tuple. This corresponds to the general structure of a URL:

```
scheme://netloc/path;parameters?query#fragment
```

Each tuple item is a string, possibly empty. The components are not broken up in smaller parts (for example, the network location is a single string), and % escapes are not expanded. The delimiters as shown above are not part of the result, except for a leading slash in the path component, which is retained if present. For example:

```
>>> from urllib.parse import urlparse
>>> o = urlparse('http://www.cwi.nl:80/%7Eguido/Python.html')
>>> o      # doctest: +NORMALIZE_WHITESPACE
ParseResult(scheme='http', netloc='www.cwi.nl:80',
path='/ %7Eguido/Python.html',
          params='', query='', fragment='')
>>> o.scheme
'http'
>>> o.port
80
>>> o.geturl()
'http://www.cwi.nl:80/%7Eguido/Python.html'
```

If the default scheme argument is specified, it gives the default addressing scheme, to be used only if the URL does not specify one. The default value for this argument is the empty string. If the allow\_fragments is false, fragment identifiers are not allowed, even if the URL;s addressing scheme normally does support them. The default value for this argument is true. The return value is actually an instance of a subclass of tuple. This class has the following additional read only convenience attributes.

Urljoin() method, used widely in this project, because constructs a full (absolute) URL by combining a base URL (href) with another URL. More specifically combining the website's url (main page) with another hrefs we have the capacity to navigate on website. Informally this uses components of the base URL, in particular the addressing scheme, the network location (part of) the path, to provide missing components in relative URL, for example:

```
>>> from urllib.parse import urljoin
>>> urljoin('http://www.cwi.nl/%7Eguido/Python.html',
'FAQ.html')
'http://www.cwi.nl/%7Eguido/FAQ.html'
```

## 2.4 Online Tools

In this section we will introduce online tools, third party APIs, such as Google Places API, Google geocoding API, The Way back Machine, Google insights etch, that have been used to develop this project

### Internet Archive

Internet Archive, also known as the “Wayback Machine” is another means of finding when a website was last updated.

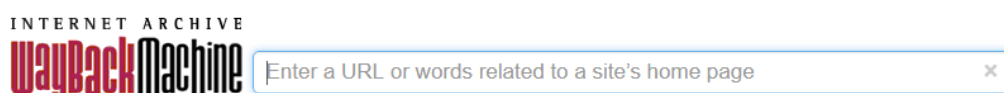


Figure 9 Internet Archive

In the search field at the top, enter the full address of the web page you want to check, including URL's scheme. Internet Archive won't give a precise date, but we can see an approximate date. Bellow the image shows the results from the Way back Machine:

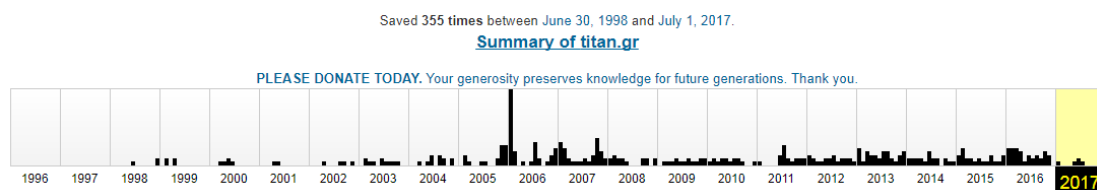


Figure 10 Internet Archive example

In this project we use Internet Archive API, to take the last modified date for websites we will crawl. More specifically we will use Wayback CDX Server API. The wayback cdx server is a standalone HTTP servlet that servers the index that the wayback machine captures. CDX API is freely available with the rest of the open source wayback machine software. The only required parameter is the URL parameter

- <http://web.archive.org/cdx/search/cdx?url=archive.org>

The above query will return a portion one per row, for each capture of the url “archive.org” that is available in the archive. The columns of each line are the fields of the cdx. At this time, the following cdx fields are publicly available:

```
["urlkey","timestamp","original","mimetype","statuscode","digest","length"]
```

It is possible to customize the Field Order as well, the url value should be URL encoded if the URL itself contains a query. The output format is JSON, output=json can be added to return results as JSON array. The JSON output currently also includes a first line, which indicates the cdx format. It is possible to customize the fields returned from the cdx server using the fl parameter. Simply pass in a comma separated list of fields and only those fields will be returned. We want the timestamp field, which is website’s last modified date according to the Internet Archive. Using the API we have the following results:

```
[["urlkey","timestamp","original","mimetype","statuscode","digest","length"],
["org,archive)/", "19970126045828", "http://www.archive.org:80/",
"text/html", "200", "Q4YULN754FHV2U6Q5JUT6Q2P57WEWNNY", "1415"],
["org,archive)/", "19971011050034", "http://www.archive.org:80/",
"text/html", "200", "XAHDNHZ5P3GSSSNJ3DME0JF7BMCCPZR3", "1402"],
["org,archive)/", "19971211122953", "http://www.archive.org:80/",
"text/html", "200", "XAHDNHZ5P3GSSSNJ3DME0JF7BMCCPZR3", "1405"],
```

## Google Places API

A way to find business location is using the Google Places API. The Google Places Web Service is a service that returns information about places – defined within this API as establishments, geographic locations, or prominent points of interest – using HTTP requests. The following place requests are available:

- Place Searches: return a list of places based on a user’s location or search string.
- Place Details requests return more detailed information about a specific place, including user reviews.
- Place Add allow you to supplement the data in Google’s Places database with data from your application.

- Place Photos gives you access to the millions of Place related photos stored in Google's Place database.
- Place Autocomplete can be used to automatically fill in the name and/or address of a place as you type.

Each of the services is accessed as an HTTP request, and returns either an JSON or XML response. The API uses a place ID to uniquely identify a place. In this project we use the Text Search service of this API, using business name.

Text Search is a web service that returns information about a set of places based on a string. The service responds with a list of places matching the text string and any location bias that has been set. The service is especially useful for making ambiguous address queries in an automated system, and non address components of the string may match businesses queries are incomplete addresses. Examples of ambiguous address queries are incomplete addresses, poorly formatted addresses, or business names. A Text Search request is an HTTP URL of the following form:

```
https://maps.googleapis.com/maps/api/place/textsearch/output?  
parameters
```

- Go to Google API console
- Create or select a project
- Click Continue to enable the API
- On the Credentials page, get an API key, set API key restrictions.

The Google Places API Web Service enforces a default limit of 1000 free requests per 24 hour period, calculated as the sum of client side and server side requests. If your application exceeds the initial limit, the app will start failing. You can increase this limit free of charge up to 150000 requests per 24 hour period by enabling billing on the Google API Console to verify your identity. A credit card is required for verification.

## Google Maps Geocoding API

Geocoding is the process of converting addresses (like “1600 Amphitheatre Parkway, Mountain View, CA”) into geographic coordinates (like latitude 37.4203021 and longitude -122.083739), which you can use to place markers on a map, or position the map. Reverse geocoding is the process of converting geographic coordinates into a human readable address. The Google Maps Geocoding API provides a direct way to access these services via an HTTP request. The following example uses the Geocoding service through the Google Maps JavaScript API to demonstrate the basic functionality.

A Google Maps Geocoding API request takes the following form:

```
https://maps.googleapis.com/maps/api/geocode/outputFormat?  
parameters
```

Where outputFormat may be either of the following values:

- JSON (recommended)
- XML

The following query contains the latitude/longitude value for a location in Brooklyn:

```
https://maps.googleapis.com/maps/api/geocode/json?latlng=40.71422  
4,-73.961452&key=YOUR_API_KEY
```

find business street address and zipcode. In case where no Google Maps is available on business website, we feed Google Places API to find business street address and zipcode if Google Places API returns None, we find business zipcode, from its website, and we use it to search business location.

To get started using Google Maps Geocoding API, we must activate an API key, so we followed these steps:

- Go to Google API console
- Create or select a project
- Click Continue to enable the API
- On the Credentials page, get an API key, set API key restrictions.

Users of the standard API:

- 2,500 free requests per day, calculated as the sum of **client-side** and server-side queries.
- 50 requests per second, calculated as the sum of **client-side** and server-side queries.

## Statsshow.com

Statsshow.com is a website analysis tool which provides vital information and estimated data of websites. Using mathematical and statistical methods, statsshow.com can estimate websites value, advertisement earnings by market niche and category and traffic such as visitors and page views. Statsshow also provides social media and internet safety reputation analysis. Below the image demonstrates the use of this online tool. As we can see we set as input the URL, from which we want to get its visits per year and its unique visits per year:

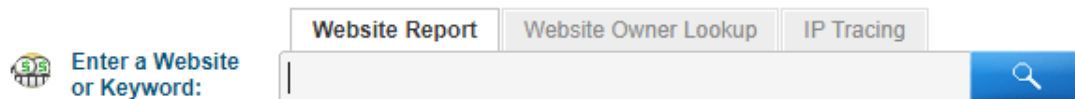


Figure 11 Statshow.com

Statshow.com has many million website reports, where elements are concerned such as daily, monthly and yearly page views, information about website's rank, websites' IP address and some other data from Alexa ranking. In this project we use this online tool, in order to be informed about the yearly page views and the unique visitors of each site. In the following capture we see the use of this online tool. Using python requests module, we grab the content of this tool in order to get the information needed (unique visitors per year and yearly pageviews).

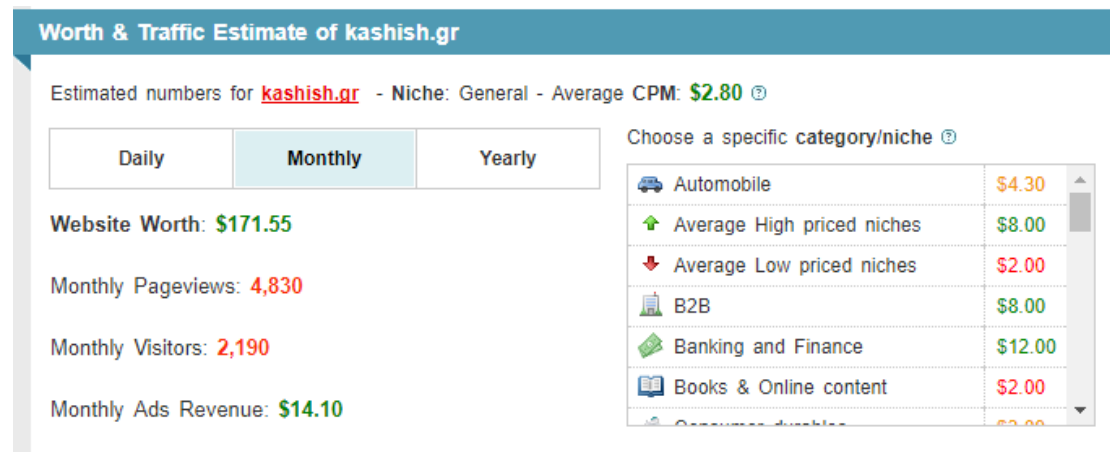


Figure 12 the Use of statshow.com

## Google Page Speed Insights API

In order to get some advanced data about the website we have, we use the Google Page Speed Insights to measure the performance of a page for mobile and desktop devices. It fetches the URL twice, once with a mobile user agent, and once with a desktop user agent. Page Speed Insights checks to see if a page has applied common performance best practices and provides a score, which ranges from 0 to 100 points, and falls into one of the three categories:

- Good: The page applies most performance best practices and should deliver a good user experience.
- Needs work: The page is missing some common performance optimizations that may result in a slow user experience.
- Poor: The page is not optimized and is likely to deliver a slow user experience.



Since the performance of a network connection varies considerably, Page Speed Insights only considers the network independent aspects of page performance, the server configuration, and the HTML structure of a page and its use of external resources such as images, JavaScript and relative performance of the page. However, the absolute performance of the page will still be dependent upon a user's network connection. In this project we use this API to measure websites' performance (for desktop), in the following capture we can see the use of this API. We set as input the website's URL and we get the following results:

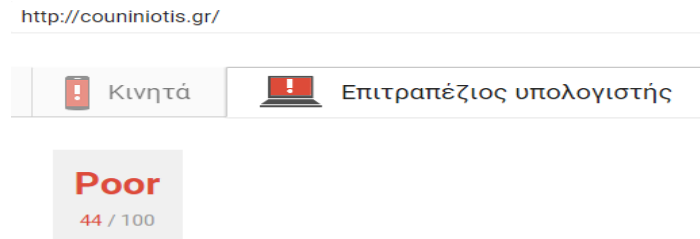


Figure 13 Google Insights API

## Chapter 3: Implementation and Results

In this chapter, we will present crawler's results and explain crawler's structure.

### 3.1 First Crawler

As it was mentioned, the crawler downloads websites' elements using the modules Requests and BeautifulSoup. The first crawler extracts social network names and URL's, and checks if the website has multi language option, search option, newsletter, blog, mobile application and e shop.

More specifically, in order to find social network names the first crawler searches for HTML elements and texts containing names of social networks such as "Facebook", "Linkedin", "Instagram" and "Youtube". As social network URL's are concerned, the crawler searches for href element, which contains social network names.

In order to find if a website provides multi language option, the crawler checks src elements, containing words such as "en", "gr", "usa", "uk", "lang" or "flag". And that's because, the language option is presented by country flags in the majority of websites.

In order to estimate if a website has newsletter, the crawler tries to locate websites' elements containing words such as "email", "newsletter", "e mail" etc. Moreover to find if a website provides search option, the crawler tries to locate elements containing words such as "search", "keyword", "form", "text box", "input" etc.

In order to check if a website has blog, the crawler tries to locates elements containing words such as "blog" or "BLOG" etc. Furthermore, in order to estimate if a mobile app is provided, the crawler searches for words like "google logo", "appstore", application etc. For e shop, the crawler tries to locate elements containing words such as "e shop", "cart", "basket", "shop" etc. The following image shows the use of this crawler:

```
In [16]: first_crawler('http://couniniotis.gr/')
Out[16]: ['http://couniniotis.gr/',
          ['Facebook', 'Twitter', 'Linkedin', 'Youtube'],
          ['http://www.linkedin.com/company/couniniotis-group',
           'http://www.youtube.com/user/CouniniotisGroup',
           'https://www.youtube.com/user/CouniniotisGroup',
           'http://wordpress.org/extend/plugins/youtube-channel-gallery/faq/',
           'http://www.facebook.com/couniniotis.gr'],
          1,
          0,
          1,
          1,
          0,
          0]
```

Figure 14 First Crawler demo

As we can see from the image above, we set as input, website's URL and then the crawler finds social network names and social network URLs. In addition, it can

prove if this website provides Multilanguage option, search option and if it has blog. In this project, we used the first crawler on 4400 websites and stored the results on csv files. In order to evaluate the results, we picked 250 random samples from these first crawler results, and we compared them with websites' raw information. For each field that the crawler extracts from these 250 random samples, we found the following:

Elements to be extracted	Correct extraction	Correct extraction (%)
Social networks (text)	222/250	88%
Social network URL	208/250	83%
Multilanguage Option (0 or 1)	228/250	91.2%
Newsletter (0 or 1)	218/250	87.2%
Search Option (0 or 1)	231/250	92.4%
Blog (0 or 1)	237/250	94.8%
Mobile app (0 or 1)	226/250	90.4%
E shop (0 or 1)	234/250	93%

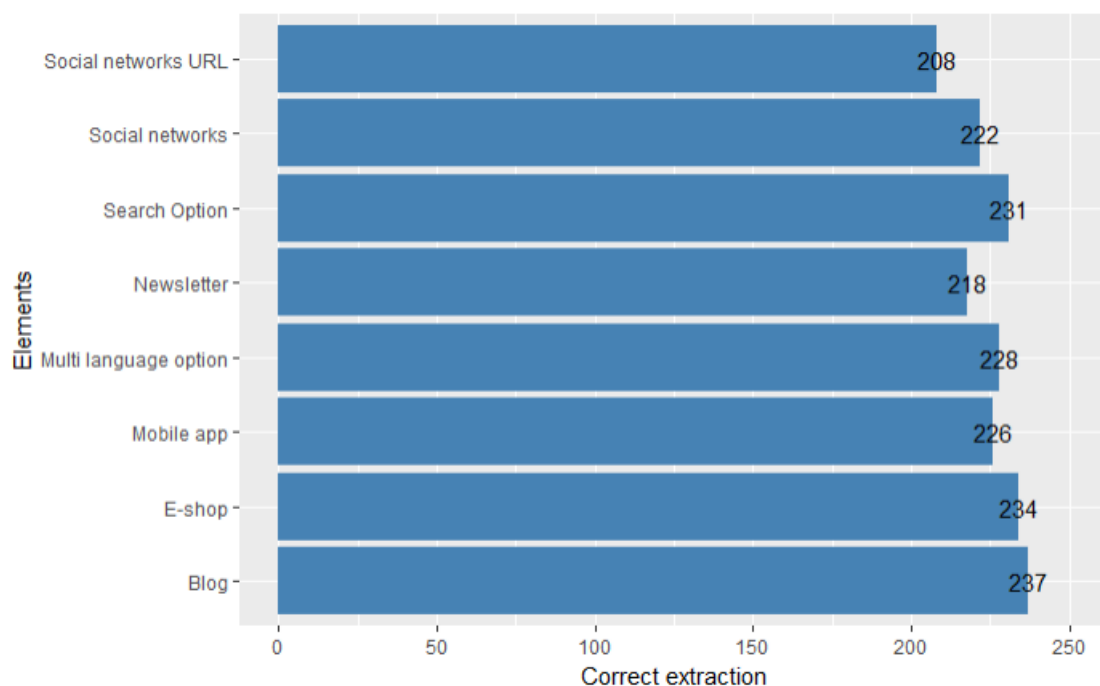


Figure 15 First crawler evaluation

As we can see from the bar chart above, the crawler is very effective to extract these elements with accuracy, which is due to the structure of the information we want to get. More specifically, the crawler checks only if these elements truly exist. Finally we conclude that 'Blog' is the most accurately located element by the first crawler. The time that the crawler needs to extract the above elements for 4400 websites, is 5117.582462310791 sec.

### 3.2 Second crawler

The second crawler consists of five functions; the first function is called “email\_crawler”. More specifically, this function searches for businesses’ emails on the website. The most difficult process was to locate emails from websites using JavaScript, in order to protect their content. We achieved this by using the Selenium Web driver, which disables JavaScript. With regular expressions, we tried to identify email addresses of the website’s text using this following regular expression:

$$[\backslash w\backslash.-]+@[\backslash w\backslash.-]+$$

The capture presents the use of the “email\_crawler” function:

```
In [6]: url1,r = check_redirects('http://couniniotis.gr/')
        email_crawler(url1,r)

Out[6]: ['eleni@kouniniotis.gr',
        'dmakris@kouniniotis.gr',
        'jmakris@kouniniotis.gr',
        'spinning@kouniniotis.gr',
        'sales@kouniniotis.gr',
        'lena@kouniniotis.gr',
        'webmaster@kouniniotis.gr',
        'warehouse@kouniniotis.gr',
        'thanasis@kouniniotis.gr',
        'info@kouniniotis.gr',
        'kkavvadias@kouniniotis.gr',
        'kmakris@kouniniotis.gr',
        'elagel@kouniniotis.gr']
```

Figure 16 Email crawler example

As we can see, we set as input the website URL and the “email\_crawler” function extracts email contacts. To achieve this, the function searches on websites, not only their main page but also internal links relating to business contacts.

The second function, which is included in the second crawler, is the “find\_certif”. This function searches for business certifications such as “ISO”, “HACCP”, “AGROCERT”, “BRC”, “OHSAS” etc. In order to find certifications, this function searches on internal links relating to businesses’ certifications, businesses’ history, businesses’ identity. In order to achieve better results, it searches on more than one links. When links are located, the “find\_certif” function searches on the websites’ text in order to identify “ISO”, “HACCP”, “AGROCERT”, “BRC”, “KIR”, “OHSAS” certifications. The following image presents the use of the find\_certif function. As we can see we set as input the website and the result is business certifications. As we notice, this function finds “ISO”, “KIR”, “BRC”, “HACCP” certifications.

```
In [3]: url1,r = check_redirects('http://couniniotis.gr/')
        find_certif(url1,r)

Out[3]: ['ISO 22000', 'ISO 22000:2005', 'ISO:22000', 'KIR', 'BRC', 'HACCP', 'CE']
```

Figure 17 Find certifications example

The third function including in the second crawler, is the “finder” function. This function searches for businesses’ phone contacts. More specifically, using regular expressions, we have model all the possible ways, phone contacts can be written. The following regular expressions identify all the possible ways:

- **[26]\d{9}**: This regular expression identifies all the possible numbers, which start with “2” or “6”, following by 9 digits such as 2691042194 or 6972915586.
- **[26]\d{4}[\s\.\-]{1,3}\d{5}**: This regular expression identifies all the possible numbers, which start with “2” or “6”, following by 4 digits included whitespace, dots or dashes, and ending with 5 digits such as: 26910 42194 or 26910 – 42194 or 26910.42194
- **[26]\d{3}[\s\.\-]{1,3}\d{6}**: This regular expression identifies all the possible numbers, which start with “2” or “6” and are formed like 2691 04294, 2691-04294, 2691.042914 etch.
- **[26]\d{2}[\s\.\-]{1,3}\d{7}**: This expression searches for sequence of digits such as, 210.9356560, 210-9356560, 210 9356560, 210 – 56560
- **[26]\d{2}[\s\.\-]{1,3}\d{4}[\s\.\-]{1,3}\d{3}**: This expression searches for contacts such as 210 9356 560, 210.9356.560, 210-9356-560 etch.
- **[26]\d{2}[\s\.\-]{1,3}\d{3}[\s\.\-]{1,3}\d{4}**: This expression searches for phone numbers such as 210.935.6560, 210-935-6560, 210 935 6560 etch.
- **[26]\d{2}[\s\.\-]{1,3}\d{2}[\s\.\-]{1,3}\d{5}**: This expression searches for phone contacts, such as 210 93 56560, 210.93.56560, 210-93-56560 etch.
- **[26]\d{2}[\s\.\-]{1,3}\d{2}[\s\.\-]{1,3}\d{2}[\s\.\-]\d{3}**: This expression searches for phones on business website such as 210 93 56 560, 210.93.56.560, 210-93-56-560 etch.
- **[26]\d{4}[\s\.\-]{1,3}\d{2}[\s\.\-]{1,3}\d{3}**: This expression searches for contacts such as: 21093 56 560, 21093.56.560, 21093-56-560 etch.
- **[26]\d{3}[\s\.\-]{1,3}\d{3}[\s\.\-]{1,3}\d{3}**: This expression searches for contacts such as 2109 356 560, 2109-356-560, 2109.356.560 etch.
- **[23]\d{3}[\s\.\-]{1,3}\d{2}[\s\.\-]{1,3}\d{4}**: This expression searches for contacts such as 2109 35 6560, 2109-35-6560, 2109.35.6560 etch
- **[23]\d{3}[\s\.\-]{1,3}\d{2}[\s\.\-]{1,3}\d{2}[\s\.\-]\d{2}**: This expression searches for phone numbers such as 2109-35-65-60, 2109.35.65.60, 2109 3 565 60

The following image demonstrates the use of “finder” function. As we can see we set as input the URL:

```
In [7]: url1,r = check_redirects('http://couniniotis.gr/')
finder(url1,r)

Out[7]: '697 667 7598,26910.74473,26910.74484'
```

Figure 18 finder function demo

The fourth function is called “find\_bname”. This function searches for business name on websites’ text. More specifically, it tries to locate capitalized words before keywords such as “A.E”, “E.Π.E”, “A.T.E”, “O.E”, “A.B.E.E”, “E.E”, “S.A”, “Co.”. In order to achieve this, we download websites’ text using the “Requests” Python module and regular expressions to identify the business name before these keywords. The regular expression we used to identify business names, is the following:

```
(([A-Ω][\w-]+[.]?|[A-Z][\w-]+[.]?)(\s(\&\s)?([A-Ω][\w-]*|[A-Z][\w-]*))*)\s([.]?K[.]?E[.]?Π[.]?E[.]?A[.]?E[.]?E[.]?LTD|[Ll]td|[A.]?E[O.]?E[.]?)
```

The image bellow demonstrates the functions’ output, when each companies name is found before these keywords:

```
In [117]: url1,r = check_redirects('http://pyrosvestires-athina.gr/')
print(business_name(url1,r))

['BAVARIA ΠΥΡΟΤΕΧΝΙΚΑΛ ΜΟΝ ΕΠΕ']
```

Figure 19 find\_bname function demo 1

In case these keywords can’t be found, the “find\_bname” function tries to locate capitalized words (Company names) after keywords such as “company”, “εταιρεία”, “οργανισμός”, “όμιλος”, “corporation”, “επιχείρηση”. The regular expression we used is the following:

```
(([οΟ]ργανισμός|[ε]ταιρεία[ς]?|company|[ε]ταιρία[ς]?|Όμιλος|όμιλος|εταιρείας μας|εταιρία|εταιρίας μας|επωνυμία|[Εε]πιχείρηση|όμιλος εταιρειών)\s((([A-Ω][\w-]+[.]?|[A-Z][\w-]+[.]?)(\s(\&\s)?([A-Ω][\w-]*|[A-Z][\w-]*))*)
```

The image shows the output, when the companies name is located after words such as, “company”, “εταιρεία”, “οργανισμός”, “όμιλος”, “corporation”, “επιχείρηση”.

```
In [118]: url1,r = check_redirects('http://www.casadino.gr/')
print(business_name(url1,r))

['εταιρεία CASADINO']
```

Figure 20 find\_bname function demo 2

In case the website refers to Hotels, the function tries to locate capitalized words (Hotel name) before keywords “Hotel”, “Resort”, “Apartments”. So we have the following output:

```
In [126]: url1,r = check_redirects('http://www.equiphotelline.eu/')
          print(business_name(url1,r))
          ['EQUIP HOTEL']
```

Figure 21 find\_bname function demo 3

In case the function can't locate these keywords and the corresponding company names, we use the title of HTML document, which usually contains business name. For example:

```
<!--[if IE]><meta
http-equiv="X-UA-Compatible" content="IE=Edge,chrome=1"/><![endif]--
<title>Perivolaropoulos - Αθηναϊκό Βιβλιοπωλείο Επιστημών</title>
```

Figure 22 HTML document title

So if we have no information about the business name from its website, we use its HTML title.. So the function returns the title:

```
In [127]: url1,r = check_redirects('https://www.perivolaropoulos.gr/')
          print(business_name(url1,r))
          Perivolaropoulos - Αθηναϊκό Βιβλιοπωλείο Επιστημών
```

Figure 23 find\_bname function demo 3

The fifth function is called, “country\_data”. This function tries to locate country names in which a company operates and its scope of activities. More specifically the function checks for country names on websites’ internal links relating to company profile or companies exports or companies imports and activities, in order to locate the countries in which a company operates and the scope of companies activities (local, national, international). The detection is done using a file in which we have stored country names, in Greek and in English, so the function “country\_data” tries to match country names from the file to websites’ text. So if country names not found, the function set the scope as local, if “Greece” or “Ελλάδα” have been found the function set the scope as national. In the case where, other country names have been found, the function sets the scope of companies activities as international. So the “country\_data” returns a list of countries in which a company operates and “0” if scope of activities is local, “1” if companies scope of activities is national and “2” if scope of activities is international. The images bellow demonstrates the use of “country\_data” function:

```
In [3]: country_data('http://www.fanis.com.gr/')
Out[3]: (['Αλβανία', 'Ιταλία', 'Γερμανία', 'Τουρκία'], 2)
```

Figure 24 country\_data demo 1

```
In [4]: country_data('http://www.e-papagrigoriou.gr/')
Out[4]: (['Ελλάδα'], 1)
```

Figure 25 country\_data demo 2

Also we must point out that, the execution time the email crawler needs to find email contacts for 4400 websites is 23907.21040248872 sec. And that's because of the Selenium Web driver, which is a very slow module. More specifically, every time Selenium is activated, JavaScript is disabled and a new pop up window is appeared.

The time needed by the find\_certif function to extract certifications for 4400 websites is 27801.28145599365 sec. This time is caused by the effort of this function's iterative operation on websites' internal links, in order to find business certifications.

The time needed by the “finder” function to extract phone contacts for 4400 websites is 3653.570245742798 sec. The time needed by the “find\_bname” to extract business names from 4400 websites was: 11929.462146759033 sec. The time needed by the “country\_data” function to extract information for 4400 websites is 5769.667720794678 sec.

In order to evaluate our results, we get 250 random samples from the second crawler's output and compare them with websites' raw information. The results are the following:

Elements to be extracted	Correct extraction	Correct extraction (%)
emails	241/250	96.4%
certifications	215/250	86%
phones	221/250	88.4%
Business name	207/250	82.8%
Countries in which a company operates	219/250	87.6%
Companies scope of activities	219/250	87.6%

As we conclude from the following chart, the second crawler locates from a website, emails more efficiently than certifications.



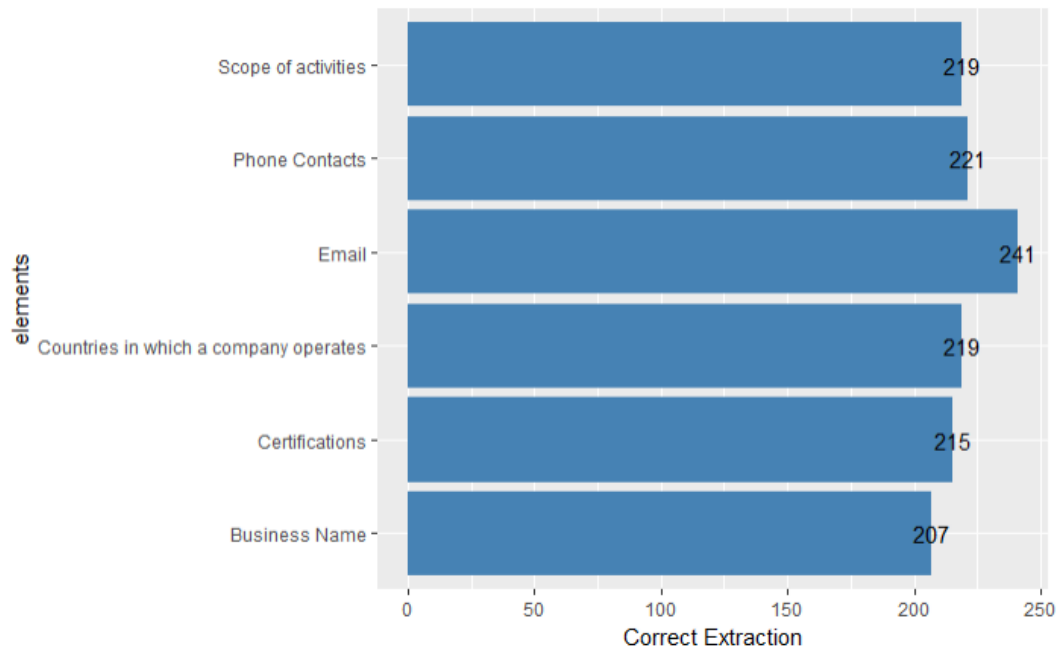


Figure 26 Second Crawler evaluation

### 3.3 Third crawler

The third crawler is consisted of 3 functions. The first function is called “website\_dev” and uses the Google Insights API in order to get information about websites’ development stats. More particularly, the Google Insights API returns percentages which represent websites’ development quality. According to Google documentation 0-65% means Poor development quality, 66%-79% Average development quality and 80%-100% Good quality. This function returns 1 for Poor, 2 for Average and 3 for Good. It also returns some other resources (CSS, JavaScript etc) providing by Google Insights API. The image bellow shows how to use “website\_dev” function. As we can see we set as input websites’ URL and we get the results.

```
In [2]: website_dev('http://couniniotis.gr/')
Out[2]: ('http://couniniotis.gr/',
         '1',
         {'cssResponseBytes': '229446',
          'htmlResponseBytes': '311383',
          'imageResponseBytes': '1981210',
          'javascriptResponseBytes': '848682',
          'numberCssResources': 16,
          'numberHosts': 6,
          'numberJsResources': 26,
          'numberResources': 128,
          'numberStaticResources': 98,
          'otherResponseBytes': '2425',
          'totalRequestBytes': '13985'})
```

Figure 27 Website Development stats

As we notice the particular website, according to Google Insights, has Poor development stats. Also we can see that the function returns information about websites' resources.

The second function is called "wayback\_machine". More specifically, this functions uses information from Internet Archive consuming its API. Using CDX API we get websites' last modified date according to Internet Archive. The image bellow, demonstrates the use of "wayback\_machine" function. As we can see we set as input websites' URL and the output is websites' last modified date.

```
In [3]: wayback_machine('http://couniniotis.gr/')
Out[3]: '26/06/2017'
```

Figure 28 wayback machine function example

The third function is called "advanced\_stats" and consumes information from Statshow.com. More particularly, we want to get websites' yearly page views and unique visits. Bellow the image demonstrates the use of this function, as we can see we set as input website's URL and we get the page views and the unique visits.

```
In [5]: advanced_stats('http://www.inventics.net')
Out[5]: ('http://www.inventics.net', '60,225', '27,375')
```

Figure 29 advanced stats function example

In order to get website development stats for 4400 websites, the "website\_dev" function needs 2390.5342102050781 sec. To find last modified date for 4400 websites, the "wayback\_machine" function needs 3153.999614715576 sec and finally in order to find total page views and unique visits the "advanced\_stats" function needs 10370.082950592041 sec.

In order to evaluate our results, we get 250 random samples from third crawler's output. Due to the fact that the collected data come from third party sources (APIs), we check if the APIs provide information about the 250 random chosen websites. Our conclusions are the following. As we notice, the crawler is very efficient to find website development stats:

Elements to be extracted	Correct extraction	Correct extraction (%)
Website development stats	250/250	100%
Website last modified date	231/250	92.4%
Unique visits per year	225/2500	90%
Total visits per year	225/250	90%

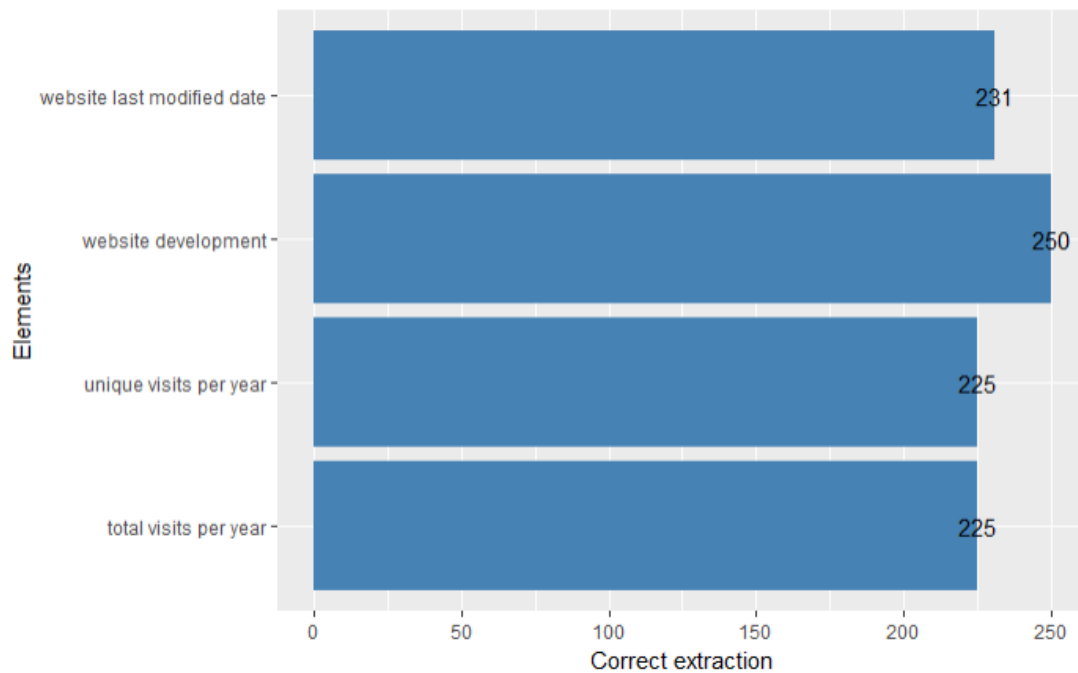


Figure 30 Second crawler results

### 3.4 Fourth crawler

The fourth crawler tries to extract elements from a website, relating to businesses geographical position such as street address, zip code and geographical coordinates. Firstly, the crawler tries to estimate, if a website has Google maps, which indicates businesses location. If the crawler finds Google maps on website, we take businesses geographical coordinates including on its source code and then we apply reverse geocoding with Google maps API or Python “Geopy” module in order to find businesses street address and zip code.

In case we cannot locate Google maps on website, we have stored on a file the Greek domain’s zip codes, so the fourth crawler tries to match every possible zip code on each website’s text. When a business zip code is found, we set it as input on Google maps API, in order to find the area in which this zip code belongs and its geographical coordinates. Also we must mention that the fourth crawler uses Selenium Web driver in order to disable JavaScript and extract geographical coordinates from Google maps source code.

The function “tk\_search” takes as input a website, then loads from a file Greek domain’s zip codes and tries to match these zip codes with websites’ text, in order to find the businesses zip code. Once a zip code is found, we set it as input on Google maps API in order to find the businesses address and its geographical coordinates. The image bellow demonstrates the use of this function. We set as input websites’ URL and the output is the businesses zip code.

```
In [22]: tk_search('http://www.pegashellas.com/')
Out[22]: 'Heraklion 713 06, Greece,35.33303980000001,25.1388411,71306'
```

Figure 31 tk\_search function

In order to evaluate function's result, we check on <http://www.pegashellas.com/> and verify the results of "tk\_search":



Figure 32 tk\_search demo

In case Google maps and zip code aren't located on businesses website, we use Google Places API in order to find businesses geographical coordinates, zip code and street address. The following image demonstrates the function of "find\_place", which uses Google Place API to find business geographical information. As we can see from the image bellow, we set as input business name and we get the following:

```
In [15]: find_place('intrakat')
Out[15]: ['19 χλμ Παιανίας - Μαρκοπούλου, Αθήνα 190 02, Greece',
          '37.942253,23.868899',
          '194 00']
```

Figure 33 Find place function

In order to verify "find\_place" result, we check on <http://www.intrakat.gr/>. We notice that Google Place API is very accurate, as the function returns the same street address with the one in the business website.

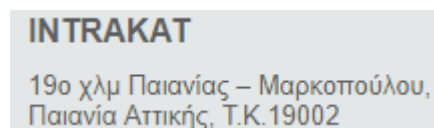


Figure 34 Find Place demo

The fourth crawler is called location crawler. The image bellow demonstrates the use of this crawler. It uses the above functions, in order to get information about business location.

```
In [31]: url1,r = check_redirects('http://www.nemavillas.com/')
          location_crawler(url1,r)
Out[31]: 'Epar.Od. Spartachoriou - Vathis, Katomeri 310 83, Greece,38.6559274717135,20.789446081984554,310 83'
```

Figure 35 Fourth crawler demo

The time fourth crawler needs to extract street address, zip code and geographical coordinates from each of 4400 websites is 48265.43081092834 sec. And that's because of the Selenium Web driver, which is a very slow module.

In order to evaluate the results, we picked 250 random samples from these fourth crawler's results, and we compared them with websites' raw information. For each field that the crawler extracts from these 250 random samples, we found the following:

Elements to be extracted	Correct extraction	Correct extraction (%)
Street address	223/250	89%
Zip code	223/250	89%
Geographical coordinates	223/250	89%

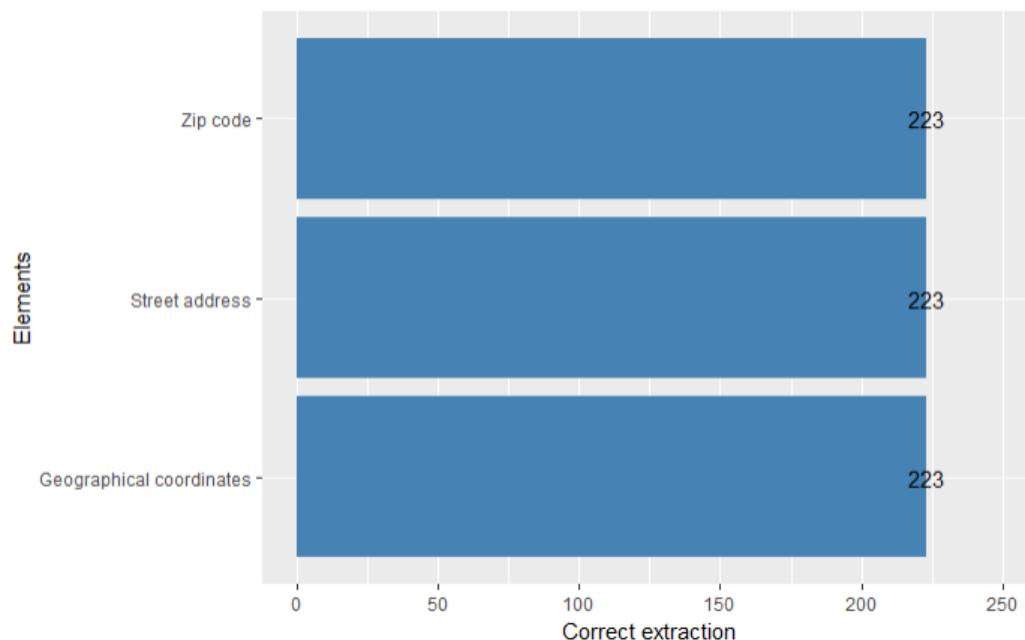


Figure 36 Fourth Crawler results

As we can see from the image above, the crawler is very efficient to locate these elements. We notice that from the 250 samples we found 223 correctly and that is due to the fact that if we have one of these elements, we can find the others using third party APIs (Google Maps, Google Places, “Geopy” module).

### 3.5 Fifth crawler

The fifth crawler is consisted of 3 functions. The first function is called “etke\_exp\_imp” and tries to locate on websites’ text terms such as “Εταιρική Κοινωνική Ευθύνη” and relating terms such as “Κοινωνική Δράση”, “Εταιρική Ευθύνη”, “ΕΚΕ”, “Εταιρική Υπευθυνότητα”. As we can see these terms are in Greek, because the majority of websites are in Greek. If the crawler locates one from these terms, returns 1 which means that on websites’ text corporate social responsibility is referred. Moreover, the crawler tries to locate the terms “εισαγωγή”,

“εξαγωγή”, their corresponding terms in English and words such as “εξάγω”, “εξαγωγικός”, “εξαγωγές”, “εξαγωγών”, “εξαγωγικός”, “εξαγωγική”, “εισάγω”, “εισαγωγικός”, “εισαγωγές”, “εισαγωγών”, “εισαγωγική”. If the crawler locates one of these words, returns 1 which means that on websites’ text the term exports or imports is referred. The images below demonstrate the use of this function. The function returns 1 or 0 if “corporate social responsibility” exists, 1 or 0 if “exports” exists, 1 or 0 if “imports” exist.

```
In [10]: url1,r = check_redirects('http://www.cavino.gr/')
         main_search(url1)
Out[10]: (0, 1, 0)
```

Figure 37 etke\_exp\_imp function demo 1

As we can see the function locates that the term “exports” is referred on websites’ text and the terms “corporate social responsibility”, “imports” is not referred.

```
In [13]: url1,r = check_redirects('http://couniniotis.gr/')
         main_search(url1)
Out[13]: (1, 1, 0)
```

Figure 38 etke\_exp\_imp function demo 2

The function locates the terms “corporate social responsibility”, “exports” on websites’ text and that the term “imports” is not referred.

```
In [15]: url1,r = check_redirects('http://www.tetramythoswines.com/el/')
         main_search(url1)
Out[15]: (0, 1, 0)
```

Figure 39 etke\_exp\_imp function demo 3

We notice that the function locates the term “exports” and not the other terms. This function needs 19230.31883239746 sec to identify if these terms exist for the 4400 websites.

The second function is called “rep\_sup\_fac” and tries to locate from websites’ text the terms “αντιπροσωποι/representants”, “υποστήριξη πελατών/customer support”, “ιδιόκτητες εγκαταστάσεις/ private facilities”. As we can see these terms are in Greek, because the majority of websites are in Greek. So the function returns 1 if “αντιπροσωποι/representants” is on websites’ text and 0 if it is not included on websites’ text, 1 if “υποστήριξη πελατών/customer support” is on websites’ text and 0 if it is not included and finally returns 1 if “ιδιόκτητες εγκαταστάσεις/ private facilities” is on websites’ text and 0 if it isn’t included. The images below presents the use of “rep\_sup\_fac” function.

```
In [4]: url1,r = check_redirects('http://www.mitsopoulos.gr/')
        main_search(url1)
Out[4]: (0, 1, 1)
```

Figure 40 rep\_sup\_fac function demo 1

As we can see, we located that on websites' text the terms “private facilities/ ιδιόκτητες εγκαταστάσεις” and “customer support/ υποστήριξη πελατών” are referred.

```
In [5]: url1,r = check_redirects('http://www.ergons.gr/gr')
        main_search(url1)
Out[5]: (0, 0, 1)
```

Figure 41 rep\_sup\_fac function demo 2

We notice that, “private facilities/ ιδιόκτητες εγκαταστάσεις” is referred on the websites' text.

```
In [6]: url1,r = check_redirects('https://www.antemisaris.gr/')
        main_search(url1)
Out[6]: (0, 1, 1)
```

Figure 42 rep\_sup\_fac function demo 3

We notice that. “private facilities” and “customer support” are referred on websites' text. This function needs 15630.31883239746 sec to locate if these terms exists for the 4400 websites.

The third function is called, find\_awards. This function tries to locate if awards are mentioned on websites' text. So returns 1 if awards found and 0 if not. For example:

```
In [4]: first_layer('http://mitsopoulos.gr/')
Out[4]: 1
```

Figure 43 find\_awards function demo

This function needs 8543.439610799154 sec to locate if these terms exists for the 4400 websites.

In order to evaluate the results, we picked 250 random samples from these fourth crawler results, and we compared them with websites' raw information. For each field that the crawler extracts from these 250 random samples, we found the following:

Elements to be extracted	Correct extraction	Correct extraction (%)
If term “exports” referred on websites	235/250	94%
If term “imports” referred on websites	240/250	96%
If term “corporate social responsibility” referred on websites	221/250	88.4%

If term “private facilities ” referred on websites	242/250	96.8%
If awards referred on websites	237/250	94.8%
If term “customer support” referred on websites	219/250	87.6%
If term “representation” referred on websites	233/250	93.2%

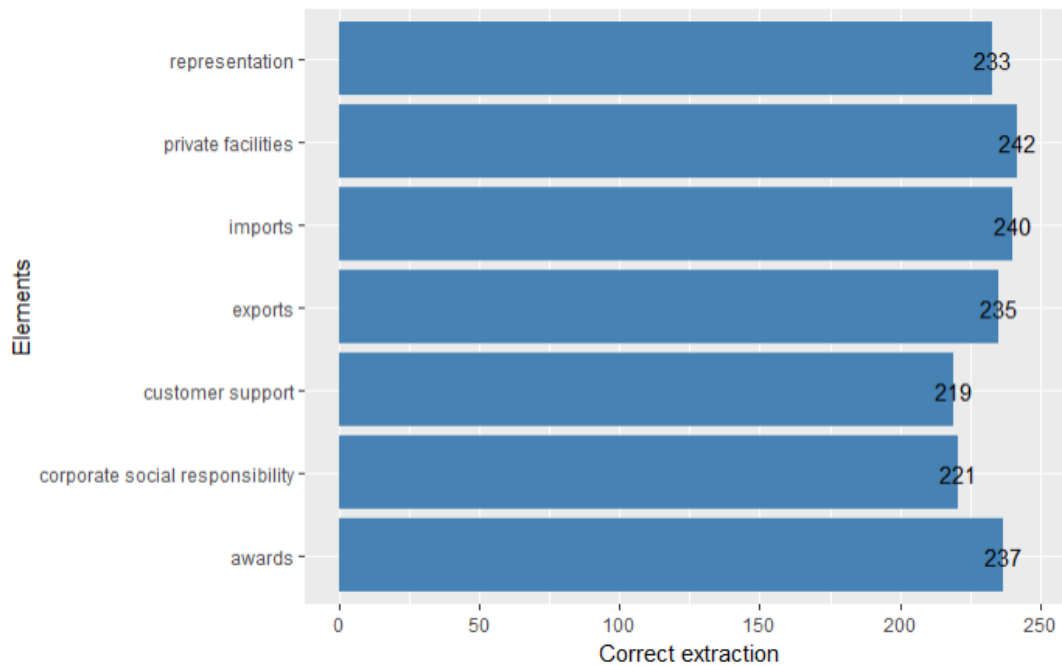


Figure 44 Fifth crawler results

### 3.6 Problems

During the implementation of this project, we faced many problems. Firstly we received net locks instead of URL addresses:

URL
<a href="http://www.debour.gr">www.debour.gr</a>
<a href="http://www.minusitsolutions.gr">www.minusitsolutions.gr</a>
<a href="http://www.service-spot.gr">www.service-spot.gr</a>
<a href="http://www.mvpsystems.gr">www.mvpsystems.gr</a>
<a href="http://www.aquasol.gr">www.aquasol.gr</a>
<a href="http://www.liebeanta.net">www.liebeanta.net</a>

Figure 45 URL form

So in order to find the full URL scheme, we create a Python script to find URL scheme (http or https). More specifically, this script tries to make requests forming URLs from net locks and http, https schemes. If we get SSL error then we change the scheme from http to https or https to http. Then we tried to find these URLs which



raise “Connection Error” or “Connection Reset Error” and these URL which their request status was 404. The URLs which raise Connection error are shown on browsers as bellow:

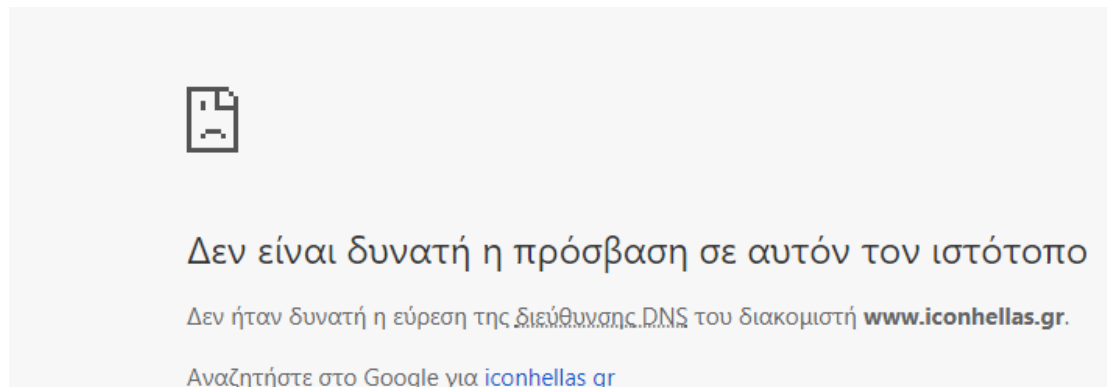


Figure 46 Connection Error

And the URLs which throw 404 status Error:



Figure 47 404 error

Another problem we faced was websites which are Adobe Flash’s applications. From these websites’ we cannot extract information, because they are applications and they are not consisted by text. These websites are presented as in the following image:

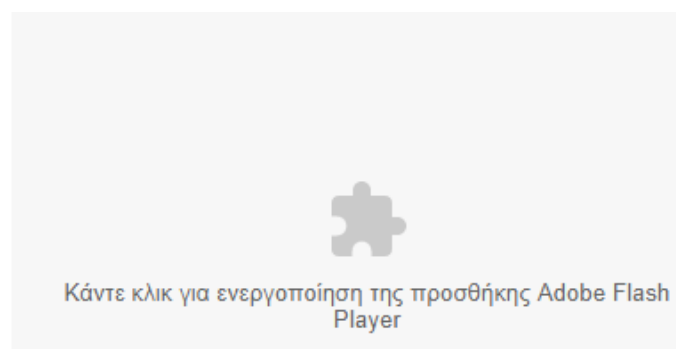


Figure 48 adobe flash applications

Moreover we had to handle websites that return HTTP 403 error. More specifically making a request on such websites; the BeautifulSoup module returns the following message:

```
In [14]: from bs4 import BeautifulSoup
soup = BeautifulSoup(r.content, 'lxml')
soup

Out[14]: <!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>403 Forbidden</title>
</head><body>
<h1>Forbidden</h1>
<p>You don't have permission to access /
on this server.<br/>
</p>
<p>Additionally, a 403 Forbidden
error was encountered while trying to use an ErrorDocument to handle the request.</p>
</body></html>
```

Figure 49 403 Error

A web server may return as HTTP 403 status in response to a request from a client for a webpage or it may indicate that the server can be reached and process the request but refuses to take any further action. HTTP status code 403 responses are result of the web server being configured to deny access to the requested resource by the client.

A common request that may result in a 403 status code is a HTTP GET request for a webpage performed by a web browser to retrieve the page for display to a user in a browser window.

In order to overcome this issue, we use user – agent software as header on Request Python module to extract the information needed. For example:

```
import requests

url = 'SOME URL'

headers = {
    'User-Agent': 'My User Agent 1.0',
    'From': 'youremail@domain.com' # This is another valid field
}

response = requests.get(url, headers=headers)
```

Figure 50 User Agents

Furthermore, one problem we faced was that some data was presented as images on websites and not as text, for example “ISO”, “TUV” certifications or phone contacts etch, making their extraction impossible.

So concluding, we mention the number of functional URLs we have in our disposal, the number of URLs which raise “Connection Error”, the number of URLs which present “403 status code” and the number of URLs which no long exists (404 status code). As we can see from the image bellow 4003 URLs are functional, 858 URLs raise “Connection error”, so we can use them, in order to extract elements, 70 URLs don’t exist anymore and 166 URLs present HTTP 403 status code which can be used to extract elements, using “User - agent” software.

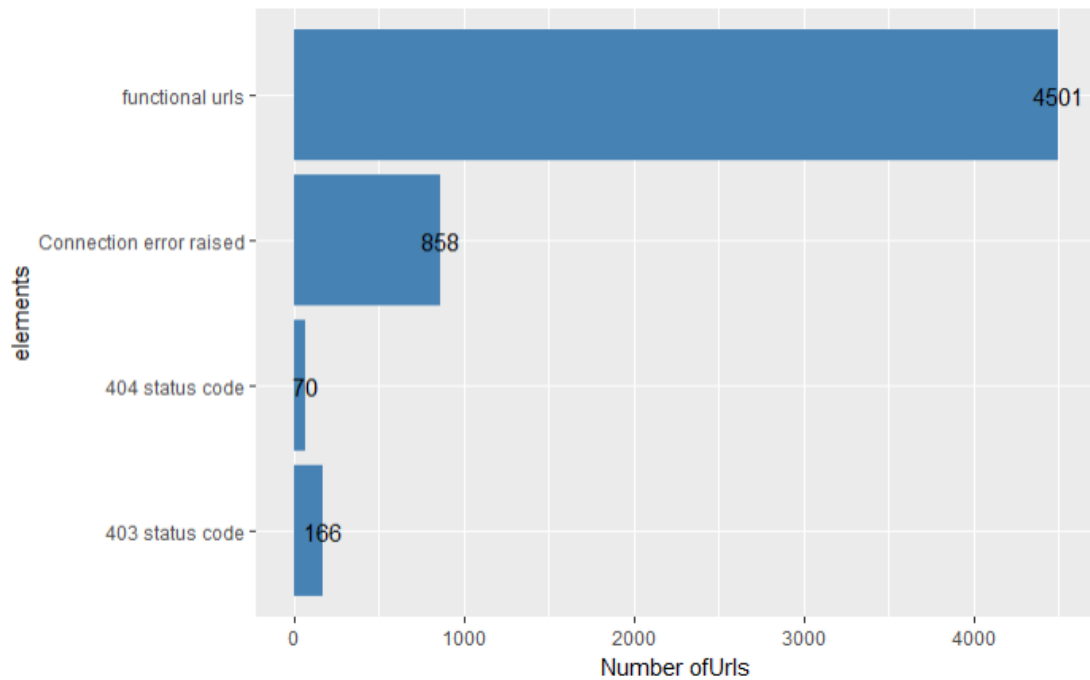


Figure 51 Urls and Status codes

### 3.7 Results

In order to evaluate our results, we picked 250 random samples from each crawler's results and we compared them with websites' raw information. For each field we found the following:

Crawlers	Elements to be extracted	Correct Extraction
<b>First Crawler</b>	<b>Social networks</b> (text)	222/250
	<b>Social networks URLs</b> (text)	208/250
	<b>Multi language option</b> (0 or 1)	228/250
	<b>Newsletter</b> (0 or 1)	218/250
	<b>Search option</b> (0 or 1)	231/250
	<b>Blog</b> (0 or 1)	237/250
	<b>Mobile application</b> (0 or 1)	226/250
	<b>E shop</b> (0 or 1)	234/250
<b>Second Crawler</b>	<b>Email</b> (text)	241/250
	<b>Certifications</b> (text)	215/250
	<b>Phone contacts</b>	221/250
	<b>Business name</b> (text)	207/250
	<b>Countries in which a company operates</b> (text)	219/250
	<b>Companies scope of activities</b> (Local or National or International)	219/250
<b>Third Crawler</b>	<b>Website Development stats</b> (poor, average, good)	250/250
	<b>Website last modified date</b>	231/250

	(date)	
	Unique visits/year	225/250
	Total visits/year	225/250
Fourth Crawler	Street address	223/250
	Zip code	223/250
	Geographical coordinates	223/250
Fifth Crawler	If exports (0 or 1)	235/250
	If “corporate social responsibility” (0 or 1)	221/250
	If “private facilities” (0 or 1)	242/250
	If “awards” (0 or 1)	237/250
	If “customer support” (0 or 1)	219/250
	If “representats” (0 or 1)	233/250

The following capture represents the success extraction for each field:

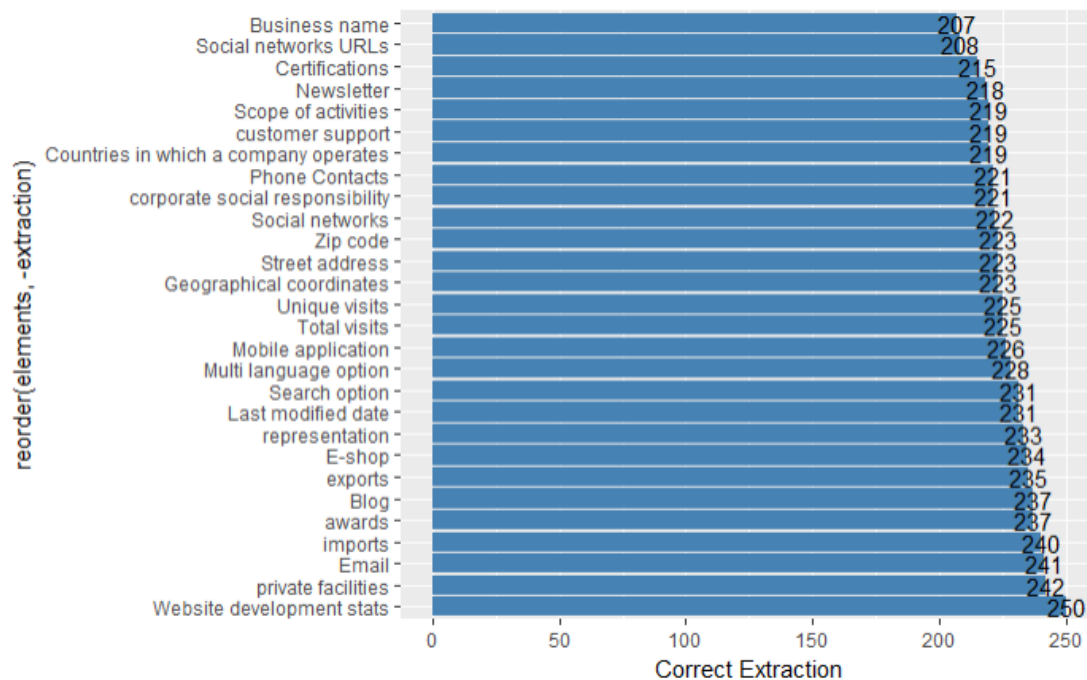


Figure 52 Correct extraction for each field

The following table demonstrates the execution time needs every function on each crawler to extract information, from the websites we have in our disposal.

Crawler	Elements to be extracted	Execution time (sec)	Execution time (Hours)
First Crawler	Social networks (text)	5117.582462310791	1.421551
	Social networks URLs (text)		
	Multi language		

	<b>option</b> (0 or 1)		
	<b>Newsletter</b> (0 or 1)		
	<b>Search option</b> (0 or 1)		
	<b>Blog</b> (0 or 1)		
	<b>Mobile application</b> (0 or 1)		
	<b>E shop</b> (0 or 1)		
<b>Second Crawler</b>	<b>Email</b> (text)	23907.21040248872	6.640892
	<b>Certifications</b> (text)	27801.28145599365	7.722578
	<b>Phone contacts</b>	3653.570245742798	1.014881
	<b>Business name</b> (text)	11929.462146759033	3.313739
	<b>Countries in which a company operates</b> (text)	5769.667720794678	1.602685
	<b>Companies scope of activities</b> (Local or National or International)		
<b>Third Crawler</b>	<b>Website Development stats</b> (poor, average, good)	2390.5342102050781	0.664037
	<b>Website last modified date</b> (date)	3153.999614715576	0.876111
	<b>Unique visits/year</b>	10370.082950592041	2.880579
	<b>Total visits/year</b>		
<b>Fourth Crawler</b>	<b>Street address</b>	48265.43081092834	13.407064
	<b>Zip code</b>		
	<b>Geographical coordinates</b>		
<b>Fifth Crawler</b>	<b>If exports</b> (0 or 1)	19230.31883239746	5.341755
	<b>If imports</b> (0 or 1)		
	<b>If “corporate social responsibility”</b> (0 or 1)		
	<b>If “private facilities”</b> (0 or 1)	15630.31883239746	4.341755
	<b>If “customer support”</b> (0 or 1)		
	<b>If “representats”</b> (0 or 1)		
	<b>If “awards”</b> (0 or 1)	8543.439610799154	2.373178

So composing the final dataset, we merge the CSV files that came up from the crawlers mentioned before, and we end up to a CSV file with 30 columns. The dataset columns described below:

#### **Dataset Columns:**

**url**: URL is the company's website for which we want to extract information, is the id element for each row. We merge the CSV files that came up from the crawlers, based on url column (id element).

**social networks**: This column contains, the Social networks which may be mentioned on company's website.

**urls from social**: This column contains, social network links, referred on company's website.

**multilang\_opt**: This column takes two values, "1" when the website has Multilanguage option and "0" when the website does not support Multilanguage option.

**newsletter**: This column takes two values, "1" when the website has newsletter and "0", when the website has not newsletter.

**search\_opt**: This field takes, "1" if the websites supports search option and "0" in the other case.

**blog**: This field takes, "1" if the websites contains blog, or "0" in the other case.

**mob\_app**: This field takes "1", if the website supports mobile application and "0" in the other case.

**eshop**: This field takes, "1" if the website contains e – shop and "0" if not.

**emails**: This field contains business email contacts.

**phones**: This field contains business phone contacts.

**bname**: This field contains business name.

**certifications**: This field contains business quality certifications.

**countries**: This field contains, the countries in which the business operates.

**range**: This field contains, the business scope of activities, "1" for local scope, "2" for national scope, "3" for international scope of activities.

**total visits**: This field contains, websites' total visits per year (source: Statsshow.com).

**unique visits**: This field contains, websites' unique visits per year (source: Statsshow.com)

**date**: This field contains websites' last modified date.

**website quality**: This field contains, website development quality, according to Google Insights API, “1” for Poor quality, “2” for Average quality and “3” for Good quality.

**address**: This field contains business street address.

**lat**: This field contains the latitude of businesses geographical location.

**lng**: This field contains the longitude of businesses geographical location.

**zipcode**: This field contains business zip code.

**if exports**: This field takes “1” if exports referred on business website, and “0” if not.

**if imports**: This field takes “1” if imports referred on business website, and “0” if not.

**if eke**: This field takes “1” if corporate social responsibility referred on business website, and “0” if not.

**private facilities**: This field takes “1” if private facilities referred on business website, and “0” if not.

**if awards**: This field takes “1” if awards referred on business website, and “0” if not.

**customer support**: This field takes “1” if customer support referred on business website, and “0” if not.

**find representants**: This field takes “1” if representation referred on business website, and “0” if not.

## References

Python Software Foundation . (n.d.). *HOWTO Fetch Internet Resources Using The urllib Package*. Retrieved from <https://docs.python.org/3.4/howto/urllib2.html>

Python Software Foundation. (n.d.). *The Python Tutorial*. Retrieved from <https://docs.python.org/3/tutorial/>

Fishkin, R. (n.d.). *How Does Google Handle CSS + Javascript "Hidden" Text?* Retrieved from <https://moz.com/blog/google-css-javascript-hidden-text>

Genesis Sample. (n.d.). *Selenium 3.5 – Complete Guide to the latest Selenium WebDriver*. Retrieved from <http://www.automationtestinghub.com/selenium-3/>

Geopy. (n.d.). Retrieved from <https://pypi.python.org/pypi/geopy>

Google maps geocoding API. (n.d.). Retrieved from <https://developers.google.com/maps/documentation/geocoding/intro>

Google. (n.d.). *PageSpeed Insights API*. Retrieved from <https://developers.google.com/speed/docs/insights/v2/getting-started>

Google. (n.d.). *Reverse Geocoding, Google Maps API*. Retrieved from <https://developers.google.com/maps/documentation/javascript/examples/geocoding-reverse>

Import.io. (n.d.). *Import.io | Extract data from the web*. Retrieved from <https://www.import.io/>

Internet Archive. (n.d.). Retrieved from <https://archive.org/web/>

Justin, D. (n.d.). *How To Crawl A Web Page with Scrapy and Python 3*. Retrieved from <https://www.digitalocean.com/community/tutorials/how-to-crawl-a-web-page-with-scrapy-and-python-3>

Michael Herman. (n.d.). *Recursively Scraping Web Pages*. Retrieved from <http://mherman.org/blog/2012/11/08/recursively-scraping-web-pages-with-scrapy/#.Wd0PRmiOOM8>

NLTK Project. (n.d.). *Natural Language Toolkit*. Retrieved from <http://www.nltk.org/>

Point, T. (n.d.). *Python Regular Expressions*. Retrieved from [https://www.tutorialspoint.com/python/python\\_reg\\_expressions.htm](https://www.tutorialspoint.com/python/python_reg_expressions.htm)

Python Software Foundation. (2017). *HTMLParser — Simple HTML and XHTML parser*. Retrieved from <https://docs.python.org/2/library/htmlparser.html>

Requests. (n.d.). *Requests: HTTP for Humans*. Retrieved from <http://docs.python-requests.org/en/master/>



Richardson, L. (2016). *Beautiful Soup Documentation*. Retrieved from <https://www.crummy.com/software/BeautifulSoup/bs4/doc/>.

SauceLabs. (n.d.). *Python Test Setup Example*. Retrieved from <https://wiki.saucelabs.com/display/DOCS/Python+Test+Setup+Example>

Scrapy. (n.d.). *Scrapy | A Fast and Powerful Scraping and Web Crawling Framework*. Retrieved from <https://scrapy.org/>

Sharma, L. (2015). *Handling of Alerts, JavaScript Alerts and PopUp Boxes*. Retrieved from <http://toolsqa.com/selenium-webdriver/handling-of-alerts-javascript-alerts-and-popup-boxes/>

StatsShow. (n.d.). Retrieved from <http://www.statshow.com/>

Urllib module. (n.d.). Retrieved from <https://docs.python.org/3/library/urllib.parse.html>

w3schools. (n.d.). *XPath Tutorial*. Retrieved from [https://www.w3schools.com/xml/xpath\\_intro.asp](https://www.w3schools.com/xml/xpath_intro.asp)

Wikipedia. (n.d.). *Reverse geocoding*. Retrieved from [https://en.wikipedia.org/wiki/Reverse\\_geocoding](https://en.wikipedia.org/wiki/Reverse_geocoding)

Wikipedia. (n.d.). *Web crawler*. Retrieved from [https://en.wikipedia.org/wiki/Web\\_crawler](https://en.wikipedia.org/wiki/Web_crawler)

Wikipedia. (n.d.). *Web scraping*. Retrieved from [https://en.wikipedia.org/wiki/Web\\_scraping](https://en.wikipedia.org/wiki/Web_scraping)

Yek, J. (n.d.). *How to scrape websites with Python and BeautifulSoup*. Retrieved from <https://medium.freecodecamp.org/how-to-scrape-websites-with-python-and-beautifulsoup-5946935d93fe>