REPORT ON DELIVERABLE D2.3

Harvesting and Transformation Service for Third-party Data

SLIPO

PROJECT NUMBER: 731581
START DATE OF PROJECT: 01/01/2017
DURATION: 36 months

Horizon 2020

SLIPO is a research project funded by European Commission's Horizon 2020 Programme.

Dissemination Level	Public
Due Date of Deliverable	Month 18 (30/06/2018)
Actual Submission Date	25/06/2018
Work Package	WP2 – POI Modelling, Identifiers, and Transformation
Tasks	Task 2.4 Harvesting and importing third data
Type	Other
Approval Status	Submitted for approval
Version	1.0
Number of Pages	40
Filename	D2.3_Harvesting_and_transformation_service_for_third-party_data.pdf

Abstract

This report presents the status of our work on POI data harvesting and transformation. We have designed and implemented a harvesting toolkit that provides functionalities for extracting and retrieving POI data from Web pages, Web APIs and data catalogues. This is complemented with a data exploration toolkit allowing to explore the data, compute statistics, identify patterns and analyse the distribution of values in POI attributes. This is essential for facilitating the user in generating insights over the harvested data, in order to appropriately steer the subsequent steps of the harvesting and integration workflow when applying third-party data. We present the functionalities provided by these tools, explain how they can be configured and used, and provide some indicative examples.

History

version	date	reason	revised by
0.1	30/05/2018	Table of Contents	Dimitris Skoutas
0.2	15/06/2018	First draft of Sections 2, 3 and 4	Pantelis Mitropoulos, Thodoris Vakkas, Dimitris Skoutas, Kostas Patroumpas, Giorgos Giannopoulos, Kai Barenscher, Pauline Baudens
0.5	17/06/2018	Multiple revisions and improvements	Pantelis Mitropoulos, Thodoris Vakkas, Dimitris Skoutas, Kostas Patroumpas, Giorgos Giannopoulos
0.6	20/06/2018	First complete draft for internal review	Pantelis Mitropoulos, Dimitris Skoutas
1.0	29/06/2018	Completed version for submission	Spiros Athanasiou

Author list

organization	name	contact information
ATHENA RC	Dimitris Skoutas	dskoutas@imis.athena-innovation.gr
ATHENA RC	Kostas Patroumpas	kpatro@imis.athena-innovation.gr
ATHENA RC	Giorgos Giannopoulos	giann@imis.athena-innovation.gr
ATHENA RC	Spiros Athanasiou	spathan@imis.athena-innovation.gr
WIGeoGIS	Kai Barenscher	kb@wigeogis.com
TomTom	Pauline Baudens	pauline.baudens@tomtom.com
GET	Pantelis Mitropoulos	pmitropoulos@getmap.gr
GET	Thodoris Vakkas	tvakkas@getmap.gr

Executive Summary

This report presents the current status of the SLIPO software for POI data harvesting and transformation. The deliverable is structured as follows:

In Section 1, we introduce the task of data harvesting and transformation for third-party data. We present an overview of the software tools we have designed and developed for this task, discussing their importance in POI data integration. Finally, we outline the challenges we faced during their implementation and the way these challenges were addressed.

In Section 2, we present our POI data harvesting toolkit. This comprises harvesters for extracting and retrieving POI data from Web pages, Web APIs, and data catalogues. We describe their features and functionalities, explain the main technical aspects involved, and discuss their general architecture.

In Section 3, we present our POI data exploration toolkit. We present its main functionalities, including the various patterns that are extracted and searched in the input data in order to analyse the values encountered in the different attributes in the POI descriptions.

In Section 4, a usage manual for the software is provided. First, the initial setup for the tools is presented and subsequently their configuration is described. Finally, we provide information about the execution of the POI data harvester and the POI data exploration tools.

Abbreviations and Acronyms

AJAX	Asynchronous JavaScript And XML
API	Application Programming Interface
CKAN	Comprehensive Knowledge Archive Network
CLI	Command Line Interface
CMS	Content Management System
CSV	Comma-Separated Values
cURL	client URL
DOM	Document Object Model
HTML	HyperText Markup Language
JSON	JavaScript Object Notation
OCM	Open Charge Map
PDE	POI Data Exploration
PDH	POI Data Harvester
PHP	PHP: Hypertext Preprocessor
POI	Point Of Interest
URL	Uniform Resource Locator
XML	eXtensible Markup Language

Table of Contents

1. Introduction

During the last years, the amount and wealth of information that is available on the Web about locations and Points of Interest (POIs) is constantly growing. This involves a large variety of diverse types of data and sources, including both authoritative sources and user-generated content, which can be downloaded from data catalogues, extracted from Web pages or queried from Web APIs. Thus, collecting and sifting through such available POI data from third-party sources is typically a tedious, time consuming and error-prone process requiring a lot of manual effort. Our goal in this task is to provide a set of tools that can assist the user in this process, reducing the manual effort involved.

Before we delve into the technical details, we explain in this section the scope of our work and give an overview of the developed tools, outlining also their importance and the challenges involved.

1.1. Scope

Our efforts in this task have focused on designing and developing two sets of tools. The first involves a framework for *POI data harvesting* (**PDH**), while the second comprises tools for *POI data exploration* (**PDE**).

In the context of the SLIPO POI data integration workflow, the purpose of PDH is to enable and facilitate the extraction and retrieval of third-party POI data, including the extraction of structured, semi-structured and unstructured POI data from heterogeneous sources and formats. The data in question concerns in principle attributes of POIs, as for example the name of a POI, the address or the coordinates, user ratings about the POI, opening hours, etc. However, the framework itself is designed to be agnostic with respect to the nature of the data so that it can be used to extract more kinds of information. This feature could be proven very useful in the extraction of additional information *indirectly* related to POIs, which could even include dynamic information, such as weather data or official documents somehow associated to a POI.

The sources of the data can vary from structured data repositories of a specific and fully defined format, to a completely unstructured Web site. Different tools of the framework apply to each case, with a specific configuration, describing the source type and format, supplied by the user. In this perspective, three different tools are considered: (i) a generic one, aiming to harvest sources lacking a specific, structured format; (ii) one intended to harvest sites offering a Web API; and (iii) one for collecting data from open data catalogues.

The PDE toolkit aims at complementing the results of the harvesting tools. It addresses data exploration, and in particular the assessment of a given harvested dataset and the provision of insights regarding the presence of outliers and other potential data quality issues. Although PDE is designed to be generic and extensible, so that it can be adapted and applied to other types of data besides POIs, specific functionalities are provided tailored to POI data. Moreover, the usage of the PDE tool is not specifically tied to the output of the harvesters, but it can be applied to any available input data.

PDE can handle two types of data, numerical and strings. In case of numerical data, standard statistical tests are performed, returning the data distribution. For strings, a basic pattern analysis is provided that can give

an overview of the dominant patterns inside the dataset, offering to the user an overview of the nature of the dataset. In addition, more specific tests can be performed when specifically dealing with certain POI attributes.

Concluding, PDH and PDE combined provide a service for harvesting and transforming POI data from third-party sources, allowing the user to assess their quality before further integration and analysis takes place within a data integration workflow. In the following, we provide a more elaborate overview of these software components.

1.1.1. POI Data Harvesting

The PDH framework consists of three separate tools, with somewhat different functionality to accommodate different types of data sources. Two of them are focused on structured data sources. Usually, the extraction of data from such sources is more straightforward. However, these sources are often dynamic, hence these tools can facilitate the data collection process by being executed periodically to maintain the input data up-to-date.

More specifically, the following tools are included:

- PDH-HTML. This addresses POI data extraction from unstructured sources, in particular, Web pages. It operates in two main stages. For a given Web site, it first generates a list with the URLs of the Web pages containing the data we are trying to collect. Then, the harvester retrieves the HTML code of each Web page, and parses it to extract the relevant POI data. For both stages, configuration provided by the user is required. Inevitably, this requires that the user has some basic knowledge about HTML in order to provide information about the HTML structure and setup of the pages to be parsed. All the required pieces of information are provided to the harvester in terms of a comprehensive configuration file in JSON format.
- PDH-API. This is used to retrieve data from sources offering an HTTP RESTful API. Although the details of each API may of course differ from case to case, we have found that the same generic pattern is observed in several POI data sources. This allows, for example, to implement a specific harvester for retrieving weather information about POIs from https://openweathermap.org, and then it is relatively easy to extend and adapt this same harvester to accommodate a different or an additional weather provider that offers a similar Web API. Typically, a weather provider offers an API accepting requests about particular places, as well as a list with the supported places and their codes. Thus, regardless of the specific details of the API, the underlying functionality of the harvester will typically involve reading an input list of POIs with their coordinates or place names, then issuing a series of requests to the API to retrieve relevant weather information for each one, and finally updating the results.
- PDH-CKAN. This addresses the case of data catalogues, in particular ones using the CKAN [CKAN]. We have chosen to focus on CKAN-based catalogues, since this covers a very large portion of the open data catalogues available online [ODM]. The harvester in this case is configured to search with tags and/or a query string, in order to retrieve data from the catalogue. It can then update any previously collected data with the newly retrieved ones.

1.1.2. POI Data Exploration

PDE assists the user in obtaining an overview and insights about an unknown harvested dataset. A dataset usually consists of several fields and each could potentially contain a huge number of different values, numeric or string. Conceptually, a dataset can be represented by a table, with a column for each field and the rows as the actual data. In our current implementation, the input to PDE is a CSV file, with the first row defining the name of the fields, and each value separated from the others with a comma.

PDE can be used to extract insights regarding the entire dataset. The user can retrieve basic information about the shape of the file (e.g., number of rows and number of columns) as well as information about the structure of the fields, e.g., which of them appear to be categorical, and in this case, which are their distinct values.

Moreover, when a specific field (column) is selected, more detailed information about the data contained in this field is computed and presented to the user. This includes several common statistical parameters of the contained values. When dealing with strings, information such as the number of missing values, the number of unique values, their minimum and maximum length, the dominant value and its frequency, are computed. Finally, for numeric types, the distribution parameters (or equivalently the distribution itself) is also computed to provide an overview of the data.

For the case of textual content, we perform a pattern analysis based on regular expressions. First, the tool attempts to identify certain patterns in the input data, and subsequently to compute their frequencies. In this way, the user is provided with a pattern distribution. Each pattern reveals information about the nature of the characters contained in a string, for example if these are Latin or Greek words, numeric and their length, symbols, mixed concatenation of characters, etc. For each pattern, an example is given to be more comprehensible. This case is considered as generic, since the tool is yet completely agnostic about the nature of the data and performs a general analysis based on these identified patterns.

In addition, in the case of categorical data (i.e., when the values of a field are drawn from a relatively small set of distinct values), the tool is able to automatically identify these categories. In fact, it is robust with respect to cases where each category may consist of several words and multiple categories are contained within each record. Once these categories have been found and extracted, their frequency distribution inside the data is calculated and provided to the user.

All functionality described so far is *completely agnostic* to the type of the specific field, which makes our software more broadly useful and generally applicable to various types of data. Nevertheless, since we focus on data concerning POI attributes, the analysis is made more specialized and optimized for field types which are frequently met.

Thus, in this specific analysis, the user can choose the type of the field under consideration. Specifically, the supported field types are the following:

- name
- address
- phone number
- price range

- rating
- opening hours.

For the first three, we apply a predefined set of patterns. Instead of constructing the patterns from the data as in the generic case, this particular set of patterns is used, and as usual their frequency and finally their distribution is calculated. These patterns have been designed to be as informative as possible for the specific field type. For the rest of the supported field types, a different approach is followed since these field types are of numeric nature. Once they have been stripped from any accompanying strings surrounding the numbers (the relative place of text and numbers, however, is taken into consideration), they are classified into appropriate bins, resulting in a distribution of the contained values.

1.2. Importance

The importance of the PDH and PDE tools is apparent in practical applications. One can use various sources of third-party POI data to update, enrich or even just confirm, existing POI data collections. A harvesting tool able to extract POI data under various formats and structures is required for this purpose. Subsequently, PDE is necessary for allowing the user to shift through this data to identify patterns, potential errors, and other useful insights to guide the next steps of the integration and analysis. For large datasets containing thousands or millions of POIs, it is clearly not feasible for the user to gain an overview about the usefulness and applicability of the data by just taking a glance at it. Hence, a data exploration tool is indispensable. The collected data often contains errors, such as missing or inconsistent values. As such, the exploration and assessment of the data can provide insights about such errors, also known as *data glitches* [DLS14], and thus guide the user to clean the data before any further processing and analysis on it is executed.

Through PDE, the user can identify potential data glitches, and then decide how to proceed. For example, the user can either clean the outliers from the data or decide that the harvesting procedure could be supplied with a more fine-tuned configuration file. Cleaning the data can take several forms. For instance, one can define a norm for the deviation of each value from what is considered as average, and based on this norm decide whether a value in the original dataset should be discarded or needs to be updated.

1.3. Challenges

1.3.1. POI Data Harvesting

Harvesting from a data repository offering an API is a relatively straightforward process. However, harvesting data from unstructured sources can be quite challenging. In this case, one needs to parse and analyse HTML documents with the desired data scattered over the entire content. Sometimes, each piece of information is enclosed in a specific tag with a descriptive attribute. Nevertheless, it is not less common that distinct pieces of information intended to be harvested (corresponding in distinct attributes) are not distinguishable in HTML terms.

In order to confront this case, one approach is to demand from the user an extremely detailed configuration file, describing precisely the HTML structure and which pieces should be harvested. Following this path is still an option when designing a tool for specialized users, even though the process of configuration would

be nonetheless time consuming. The approach which was chosen, however, follows a different path. Specifically, the HTML structure of the web page is treated as a Document Object Model (DOM). The user has to provide just one characteristic for each piece of information that she wants to harvest. This characteristic could be the id of the parent HTML element or any other unique attribute, but it could also be a constant text from the visible HTML page, for example a title.

Another challenge that had to be confronted during the Web harvester tool design was a way to obtain a list of all the Web pages (URLs) containing the content to be harvested. In practice, harvesting is a two-stage process. In the first stage, it harvests the URLs (supplied with a similar configuration file), while in the second stage the actual harvesting happens, locating and extracting the content of each URL harvested in the previous stage.

1.3.2. POI Data Exploration

One common issue in data exploration is the determination of outliers [DFMZ18]. An outlier is defined as an observation that lies an abnormal distance from other values in a random sample from a population. Outliers are not necessarily abnormal values. It is up to the analyst to decide whether to reject or not the outliers. In any case, outliers contain valuable information about the process of data collection, and it is important to understand the reason of their appearance.

When dealing with harvested data, or in general with data containing different types of POI attributes, there is no standard way on defining a distance between values in a population sample. This is the main challenge the current tool had to confront, and at the same time its main goal. Instead of explicitly identifying the outliers themselves, the tool constructs a pattern distribution. By visualizing this distribution, the analyst can then figure out the alienated patterns, and decide whether these patterns match with the specific kind of data or not.

Another main challenge in the process of exploration was the case of categorical fields. It is very common that each value contains more than one categories. This is not a problem as long as the categories are explicitly separated. However, in the absence of a delimiter (which may be missing due to errors in harvesting or because it is indeed missing in the original data), the categories need to be determined automatically. One option is to check the frequency of a particular sequence of words. If a sequence appears always unseparated, then it is likely that it represents a single category. It is possible, however, although very unlikely, that two categories always appear together, and with the exact same order, thus misleading the tool to recognize this pair of categories as a single category.

2. POI Data Harvesting

In this section, we describe our POI data harvesting toolkit, giving some technical insights. We present the types of data sources that are supported, and outline its current features and functionalities. We also present the software's architecture and configuration, giving some corresponding examples.

2.1. Features and Functionality

Three different harvesters are provided, as outlined in Section 1.1.1, namely PDH-HTML, PDH-API and PDH-CKAN. Each one is responsible for harvesting a specific type of data source, thus jointly covering a large number of data sources existing on the Web. PDH-HTML can be used to extract information from Web pages. PDH-API can be used to retrieve data from sources offering a RESTful Web API. Finally, PDH-CKAN can be used to collected data from open data catalogues operating on the CKAN platform.

In all cases, the results are stored in an output file in JSON format or, for the case of PDH-HTML, in CSV format. The user can also choose whether to include all the results or to write only the updates with respect to a previous run.

The source code for PDH is available at the GitHub repository of the SLIPO project[1] and is available with an Apache License v2.0. In the following, we describe each harvester in more detail.

2.1.1. HTML Harvester

PDH-HTML is designed to extract POI data from Web pages by parsing and processing the HTML code of the page, given instructions provided in an accompanying configuration file. It operates in two steps. First, it fetches a list containing the URLs of the Web pages of interest (e.g., obtained by parsing a page within the Web site that lists all available POIs of a certain category). Then, it parses and extracts the content of each page asynchronously.

Before we proceed into describing the two steps separately, we provide some details about the harvesting procedure itself. This procedure, although it presents some variations during the two steps, it has many common features.

First, we assume that the Web site to be harvested contains several HTML pages with a similar structure. Note that this is typically the case in practice. Inside these pages there exist certain elements with the information we would like to collect. The task is to retrieve these pieces of information in a systematic way and assign each one of them to a specific field. In order to achieve our goal, we have to identify the common pieces among all the structures which are associated with the information we are searching for. This procedure, which has to be manually performed by the user, produces the content of the configuration file provided to the harvester. This content can be logically separated into two parts, corresponding to the two steps of the harvesting process.

[1] https://github.com/SLIPO-EU/poi-data-harvesting

For the first step, the one associated with retrieving the list of page URLs, the most essential element is a starting point, which should be provided by the user. The starting point is a URL from which the harvester will start searching. This initial step yields a list of links, which may be either the final list or an intermediate list. In the latter case, the procedure iterates itself, taking each link as the new starting point, until it finally reaches the complete list with the links of the content that is going to be harvested.

In practice, given a starting point, two pathways can be followed. The harvester requests the starting point URL and tests the response to determine whether it is a JSON string or not. The technique used in the two cases is different. In case it is a JSON string, retrieved by an AJAX request, the response is transformed into an object and is searched by its keys. In the second case, the response is HTML content and HTML harvesting takes place. In both cases, the results are saved as an object and the user has the option either to stop the process at this point, obtaining this list which would be written in a JSON file, or to proceed to the second step, the actual process of POI data extraction. For the former case, the output file can be the input for future harvesting, without having to retrieve each time the URL list.

Once the list is granted, either obtained in a previous harvesting step or given directly in a JSON file, the harvester starts requesting the URLs of the list. The requests occur simultaneously, with the maximum number of simultaneous connections defined by the user. Each time a response is obtained, the HTML content is transformed into a DOM object in order to make parsing and searching easier. The harvester searches the DOM for tags, which have been defined by the user in the configuration file, and it assigns the values it finds to the corresponding fields. These tags could be, for example, titles in the text, the common pieces across all the web pages mentioned previously, or an attribute of an HTML tag (e.g., the id, the class or any other attribute).

2.1.2. API Harvester

PDH-API covers a common functionality met on various APIs on the Web. The APIs under discussion have their data structured in a specific way. Each dataset is located behind a specific key (id), so that one could get results in case a key is provided along with the API request. The functionality of this tool shows some similarity with the HTML harvester, in the sense that its first task is to create a list with the URLs corresponding to the total set of keys. This is usually achieved by a specific request prescribed by the user.

When this first step is completed, the harvester proceeds to the second step which is the actual retrieval of the data. In contrast to the HTML harvester, in this second step there are two options. Either it proceeds as usual by iterating over all the URLs of the constructed list, or it executes a query over the list and provides only the relevant results. For example, this harvester can be used to fetch weather data. The keys in this case correspond to the city codes. The user may be interested in specific cities rather than the entire set of results. Moreover, the weather provider may have imposed limits on the number of requests offered free of charge.

2.1.3. CKAN Harvester

PDH-CKAN can collect data from CKAN-based data catalogues. CKAN is an open source, web-based management system for open data distribution. CKAN is *the de facto* open data catalogue, in use from the majority of open data catalogues worldwide, justifying the specialized support we provide.

CKAN stores data separated in datasets and resources. A dataset typically comprises two parts. The first is information about the data, known as metadata. Metadata could be, for example, the format of the actual data, the publishing authority or the type of licence under which the data is published. The second is the data itself. The data is stored inside resources. A resource could be, for example, a file or a Web page. The format of the resource is free, it could be a JSON or GeoJSON file or even a flat CSV. Additionally, the resource could be part of the repository system or it could be provided by (and stored in) external sources.

Each dataset may be associated with a specific tag, while each tag may be assigned in multiple datasets. Moreover, a vocabulary is a collection of tags. As a result, each tag may be either a free tag or a tag contained in a vocabulary.

One option offered to the user is to harvest every single dataset contained in a CKAN catalogue. In this case, the application, as in the case of HTML harvesting, proceeds in the two familiar steps. First, it requests a list with the total number of packages (datasets). The CKAN API responds to such requests with a list containing the ids (actually the names, the two terms are used interchangeably by CKAN in this case) of the datasets. Once the list has been fetched, multiple asynchronous requests are issued to fetch the actual dataset corresponding to each item in the list.

Alternatively, the user can issue queries on the datasets. There are two ways of searching inside a CKAN catalogue. One is by searching among the tags attached to each dataset (which could be optionally grouped in vocabularies). The second is by issuing a query over all of the available fields of the datasets. Obviously, one could combine the two methods, e.g., to query the datasets with a free text and filter the query with specific tags (and/or vocabularies).

2.2. Implementation

PDH has been developed in PHP, around two fundamental tools. One of them is the cURL project[2], which provides a library for transferring data using various protocols. The other is the DOM extension[3] of PHP, which provides the DOMDocument class allowing to operate on XML documents through the DOM API with PHP. Both tools are essential for the functionality of the harvester. More specifically, cURL is used by all the three harvesting tools, while the DOM extension is required only by the HTML harvester.

To increase the functionality and usability of these tools for the specific needs of the harvesting procedure, two custom classes have been developed. CurlTools provides a couple of simple, yet useful tools for data transferring. One offers the default XML connection options, while the other simplifies the connection with several URLs asynchronously. At the same time, it offers the possibility to monitor the memory usage in order to limit the simultaneous connections if a certain threshold is exceeded (since the processing occurs simultaneously).

The cURL library has the benefit of using a callback function for requests, which is called each time a request has been fetched. This way, the script is being processed in parallel, decreasing significantly the time needed

[2] https://curl.haxx.se/

[3] http://php.net/manual/en/book.dom.php

to harvest a whole Web site, although the harvesting time is of course strongly dependent also on the network connection.

DOMTools is a class that offers tools to construct the DOM structure of the HTML page and query for keywords in the tree, as well as to create the list containing the URLs to be harvested and ultimately a tool which provides the actual harvesting from an HTML structure. Optionally, before constructing the XML structure, DOMTools use the tidy PHP extension, which is a binding for the HTML Tidy[4] clean and repair utility. It is called only in the case the extension is available, in order to provide with a correct HTML code, although it is not essential in the workflow (minor errors in the HTML code do not affect the procedure).

For searching efficiently inside the HTML structure, the HTML code is transformed into an XML DOM tree and then the XPATH query language is used [XPATH]. XPATH provides the ability to navigate around the tree representation of the XML document, selecting nodes using various criteria. In practice, when using the HTML harvester, one provides the script with keywords for which the HTML code of the Web pages would be searched for. These keywords specify the node selection criteria, searching for ids or class XML attributes (standard HTML attributes) as well as plain text.

The procedure described above is performed by the method findTag of the DOMTools class. This method is used to provide with an array containing all nodes with the requested characteristics. The harvestData method, on the other hand, iterates over the nodes of the structure as well as their children in order to find the data contained in the requested tags. Finally, constructList iterates in order to extract the href attribute and create the URL list.

The general architecture of the three different harvesters is essentially the same. They are built around a main class, which extends a parent class called Harvester. Since the procedure followed by the three tools is quite different, the functionality of the main class is limited to basic aspects, such as parsing the input arguments, opening the appropriate files and writing the output in the requested format. CurlTools is as well initialized in the parent class and it is saved in a public variable available to its children.

The CKAN and API harvesters utilize only CurlTools since they only need to call an API service. Their main classes, CKANapi and ApiHarvester respectively, are designed to take advantage of the corresponding API functionality. Since the structure of these tools is quite straightforward, we are going to focus below on HTML harvesting. However, in all cases there are two common methods. One is collect and the other is fetchData. Even though for each case the implementation is somewhat different, the former is used to retrieve the list of URLs, while the latter is the actual data extraction method.

For HTML harvesting, two classes have been implemented. One of them, the LinksList, is responsible for fetching the list of URLs (in case no AJAX functionality is provided), while the other, the WebHarvester, performs the actual harvesting on HTML pages, after possibly creating a list of links in case it is provided in JSON format. Although the main functionality for HTML harvesting is provided by DOMTools,the purpose of the WebHarvester is to provide some useful tools, covering the details. For example, WebHarvester provides a method for extracting coordinates from a Web page, using regular expressions, as well as a method to detect and fix the Web page encoding.

[4] http://www.html-tidy.org/

The main methods of WebHarvester, however, are as in all cases two. The collect method collects all the URLs from a Web site according to some predefined criteria. These criteria are the configuration provided to the script, resulting in the URL list to be harvested. The second method is called fetchData, which does the actual job of data extraction. The output of these tools can be either a CSV file or a JSON document.

Although both methods for list retrieval and data extraction employ similar XPath queries from DOMTools (with the exception of AJAX requests), there is one major difference in their operation. Collect fetches the URL content one by one, while the harvesting works asynchronously. This choice is inevitable, since, for the case of URL collection, the URLs that follow have to be retrieved from the currently requested content. Once the list is constructed, the harvester could in bulk target on the URLs and simultaneously process their content.

The harvesting toolkit is completed with a PHP wrapper script as an example of applying the tools described so far, after selecting the appropriate harvesting method (given by the user). For example, for the case of HTML harvesting, the script reads a configuration file in JSON format, which contains all the information required to create the URL list as well as those pieces of information required to harvest appropriately these URLs. As expected, this script runs in two rounds. In the first one it constructs the list, whereas in the second it feeds the list to the actual harvesting procedure. However, it is possible to adjust the script accordingly. If, for example, the URL list is already provided, the first round is omitted.

3. POI Data Exploration

In this section, we describe the POI data exploration toolkit in more detail. First, we provide an overview of its architecture, emphasizing on a technical description of its parts that could also be reused independently or integrated in other applications. Next, we give a more detailed analysis of each component, starting with the description of pattern and category recognition procedures, followed by the description of the statistical analysis corresponding to each field type. We conclude by giving some examples using these components and demonstrating their functionality.

The source code for PDE is available at the GitHub repository of the SLIPO project[5] and is available under the Apache License v2.0. In the following, we describe each harvester in more detail.

3.1. Implementation

The PDE tool has been developed in Python 3. Python is a widely used interpreted programming language, with rich open source library support, especially for applications focusing on numerical analysis and statistics. Version 3 presents many improvements over the previous major version 2. An important one for this project is the native support for UTF data instead of ASCII of version 2. Additionally, the lack of libraries ported to the new version has significantly improved recently.

Further, we exploit a number of third-party libraries. Specifically, we take advantage of the libraries numpy[6] and pandas[7], which are very commonly used in projects dealing with statistics. Python does not natively support array data structures. Numpy is a library that supplements the programming language with the feature of large, multidimensional array data structures, along with a rich ensemble of mathematical functions. Pandas is a software library focusing on data manipulation and analysis. It extends the numpy arrays to the structure of dataframes with integrated indexing. It also supplies its dataframes and series objects with several methods for analysing their data.

Along with these libraries, the tool also takes advantage of some standard third-party modules. It uses the re module to compile and search for regular expressions, similar to those found in Perl. Additionally, it uses the functionality offered by OrderedDict of collections, which is an alternative to Python dictionaries and, also, the unicodedata to manipulate strings in a language independent way. Finally, the json module is used to transform python dictionaries to JSON format.

---

[5] https://github.com/SLIPO-EU/poi-data-exploration

[6] http://www.numpy.org/

[7] https://pandas.pydata.org/

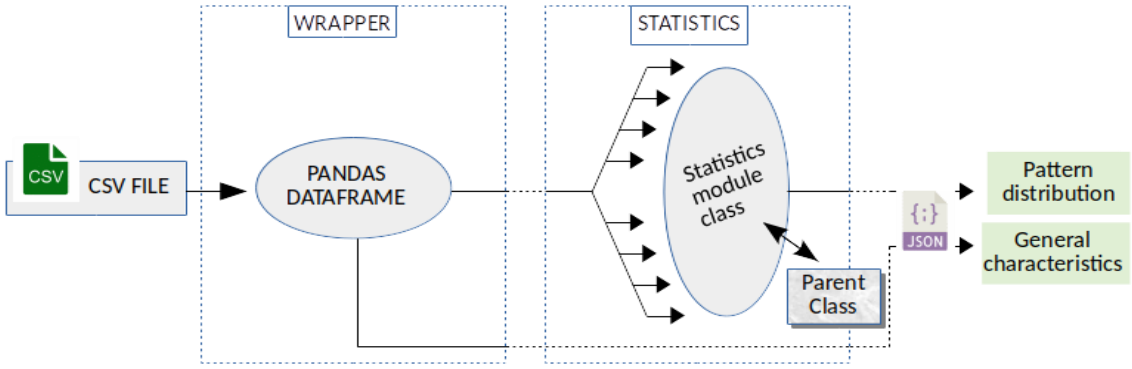


Figure 1: PDE processing flow.

The processing workflow of the PDE toolkit is shown in Figure 1. PDE essentially consists of a Python package called statistics. This package is a collection of eight modules. Each module is a class adding a certain functionality to the parent class of the package. The modules correspond to the categories of the data fields, each performing the respective tests. PDE has been supplemented with a wrapper module, which can also be used as a standalone script.

Each module can be initialized with a pandas series, which typically represents the data of a specific field (i.e., the data collected for a particular attribute for the POIs). During the initialization, all the required methods are called to run the appropriate tests, producing a public dictionary with the final data. This dictionary is accessible from the wrapper which transforms it to JSON and saves it as a JSON file, suitable to directly transform the data into a chart (typically a pie or a bar chart).

Apart from the specific results, the wrapper class offers an independent functionality. The statistics package is dedicated to the analysis of each field separately. The wrapper class prepares the data for analysis, while also offering methods to extract general statistical information about the whole dataset. Briefly, the wrapper class can give the shape of the dataset and, additionally, its structure with the categorical fields and their unique values. This material is, as usual, transformed into JSON and written in another output file, in case the wrapper class is called from the default embedded script.

3.2. Datasets and Field Types

The actual input to the PDE toolkit is a pandas dataframe object. However, for a more user-friendly approach, the script asks the user to provide the dataset in the format of a CSV file. In any case, the dataset in either format consists of a number of columns, with each column comprising data for each field (i.e., attribute) of the entity under investigation. These columns of the dataset are internally represented by pandas series objects.

The pandas series object is passed to the appropriate module in order to analyse it. The module selection is accomplished according to the type of the field. A field type may fall into a predefined category or not. In

the former case, certain predefined patterns are provided and are searched for in the values of this field to find matches. In the latter case, we attempt to identify whether the values of this field appear to be arbitrary or drawn from a set of distinct values (e.g., POI categories).

3.2.1. Identifying Patterns and Categories

Patterns are identified by the module generic of the statistics package. In this module, an iteration over the data series takes place. For each string, a pattern is constructed according to some predefined character groups (character sets, symbols, numeric values, etc.). The patterns, afterwards, are compiled into groups with specific sequence and, given this schema, a corresponding regular expression is generated.

Once we have a regular expression matching the specific string, the entire series is tested against this expression. This test is performed using the pandas regular expression toolkit. The frequency of the regular expression appearance is saved under a user-friendly name to describe the pattern, along with the initial string as a representative example of the pattern. Since the pattern of these strings has been identified, they are removed from the series, and the procedure continues with the next non-matching string.

In case the field is of categorical type, one expects each entry to consist of a sequence of words. Groups of them represent the categories of this field. Even though the number of combinations of the existing categories may be very large, the number of the categories themselves can be sufficiently manipulated. The task of the categorical module is to extract the set of the distinct categories and create the appearance distribution.

In this module, a new series is constructed containing all the words of the original series, one in each entry. The module iterates over the words and registers each word into a separate list. Each time the appearance frequency of the sequential appearance of this list items is calculated. In case a frequency is not equal to the previous value, it means that the previous sequence constituted a category. Otherwise, the procedure is continued to test whether the current sequence combination is a category. In this manner, it is feasible to find all the unique categories of the data, in a reasonable time, even if there is no delimiter to separate them.

3.2.2. Statistical Analysis of POI Attributes

Currently, six POI attributes are treated explicitly. Three of them check the ensemble against some predefined patterns, while the rest three transform the entries to numbers, removing the text, although keeping useful information about its position inside the entry. It is important to note, that even though the list of the predefined pattern types has only three representatives, it is straightforward to be extended with new members by defining extra patterns. The specific POI attributes that are handled separately, and the predefined patterns used for each one, are presented in the following and are also listed in Table 1.

- *Name*. The name is usually unique for each POI (although not necessarily). Clearly, an analysis of distinct values is not meaningful here. Moreover, it is possible that any character is allowed inside a name. With these remarks in mind, we conclude that one is usually interested to know the alphabet in which the name is written (e.g., Latin versus Greek characters), the number of words, and any symbols that it potentially contains. As a result, the patterns that are tested inside the Name module include language character sets and symbols. The information of the number of words is also extracted.

- *Address*. The scheme or writing style of addresses often varies. We try to capture various ways of writing an address by compiling a set of predefined regular expressions in the Address module. Special complication emerged for the postal code, since in some countries it may be a 3-digit number, which could also in principle represent the street number. However, taking into account its relative position inside an address string, we believe the module usually makes the correct distinction among them. Concluding, the module tests all the known address patterns, returning their frequency, as well the frequency of remaining cases, i.e., those not matching any of the current patterns. This latter case may represent wrong data or a malformed entry.
- *Phone number*. The writing of a phone number may also vary, but it is possible to identify some common patterns. Notably, it is expected to contain a series of numbers, and may also contain plus (+) or minus (-) signs, or parentheses. The patterns containing combinations of these characters are checked in the corresponding module PhoneNumber, as well as a pattern containing no number at all. This latter case could be considered almost safely as wrong. All remaining cases are returned as being unrecognized, leaving the user to decide about their legitimacy.
- *Price range*. Usually, the price range is given either in the form of two values, which define the range, or in the form of one value, defining the lower or upper limit of the range. Before, between and/or after the numbers, it is very likely to have text or symbols. Since we have to be language agnostic, we have to use some conventions. Thus, if there is text before one single value, we assume this value to denote the upper limit of a range. A common example is phrases such as "up to 20€", which are treated as denoting the range from 0 to 20. On the other hand, if text appears after a single value, this value is considered to represent a lower limit. Of course, these assumptions can be configured and adapted by the user, to fit different scenarios. This procedure is implemented in the PriceRange module, which finally constructs the appropriate bins and distributes the values in them.
- *Rating*. A rating usually is, or can be represented by, a single number. In case there is also text inside the entries, it is omitted. In the rating module, the ratings are distributed to bins, which are automatically created according to the found values.
- *Opening hours*. This field usually presents quite high diversity in its form. This makes it even harder to follow a language-independent approach in this case. However, since day names definition is not difficult in the languages we are interested in, this is not considered as a problem. Actually, what we focus on is not only the name of the days, but also their order inside the week, since for a given format, the missing, in between days of a range (e.g., in phrases such as "Mon. – Fri."), have to be filled in. Inside the Schedule module, the series is turned to lowercase, and all accents are removed. Subsequently, the dominant locale is determined in order to choose the set of day names to use. Once we are done with the locale, the series are split by the opening hours range, using the appropriate regular expression. In this way, a distribution with days and opening hours is constructed.

Table 1: List of predefined regular expressions used to identify patters in the values of specific POI attributes.

Category	Regular expression	Description
Name	(?:[ς ερτυθιοπασδφγηξκλζχψωβνμΕΡΤΥΘΙΟΠΑΣΔΦΓΗΞΚΛΖΧΨΩΒΝΜέύίόάήϋΰϊΐΈΎΊΌΆΉΫΪ0-9]+(?:$|\ ))	Greek characters
	(?:[a-zA-Z0-9]+(?:$|\ ))	Latin characters
	(?:[^\(^\[^\]^\ )]+(?:$|\ ))	Other or mixed language
Address	(?:(?:(?:[^0-9]*[0-9]+[^0-9]{1,2}\s[^0-9]+)+)|(?:[^0-9]+))(?=\s|$)	Words regular expression (language independent), e.g. street name, city etc.
	(?:[0-9\-]+[ABCDabcdΑΒΓΔαβγδ]{0,1})	Street number
	(?:\d{5}([\-]\d{4})?)|(?:[0-9]{4})|(?:[0-9]{5}[\-]?[0-9]{3})|(?:[A-Za-z][0-9][A-Za-z] [0-9][A-Za-z][0-9])|(?:[0-9]{3,4})|(?:[1-9][0-9]{3}\s?[a-zA-Z]{2})|(?:[0-9]{5})|(?:\d{3}-\d{4})|(?:(L\s*(-|—|–))\s*?[\d]{4})|(?:[0-9]{2}\-[0-9]{3})|(?:((0[1-9]|5[0-2])|[1-4][0-9])[0-9]{3})|(?:\d{3}\s?\d{2})|(?:[A-Za-z]{1,2}[0-9Rr][0-9A-Za-z]? [0-9][ABD-HJLNP-UW-Zabd-hjlnp-uw-z]{2})	Postal code
	(?:\s+)	A regular expression for space(s)
	[\-\`\~\!\@\#\$\%\^\&*\(\)_\+\[\]\{\}\;\:\"\'\\\,\.\<\>\/\?€]	Symbols
Phone number	^[0-9]+$	Only numbers
	^(?:[0-9]*[\(][0-9]+[\)][0-9]*)+$	Numbers and parentheses
	^(?:[0-9]*[\+][0-9]+)+$	Numbers and + symbol
	^(?:[0-9]*[\-][0-9]+)+$	Numbers and - symbol
	^(?:[0-9]*[+][0-9]+[-][0-9]*)+$	Numbers, + and - symbol
	^[0-9\(\)\+]+$	Numbers, parentheses and + symbol
	^[0-9\(\)\-]+$	Numbers, parentheses and - symbol
	^[0-9\(\)\+\-]+$	Numbers, parentheses, + and - symbol
	^[^0-9]+$	Only non numerical characters

4. Usage Manual

In this section, we provide a usage manual for the user of the PDH and PDE toolkits. In the following subsection, we give the details about the minimum setup required for running the tools. Next, we examine the configuration of the tools before execution. Although data exploration has rather straightforward configuration, harvesting is a procedure that requires non-trivial input by the user. Finally, we take a look at the execution of both PDH and PDE, presenting results of specific examples.

4.1. Initial Setup

Both tools can be downloaded from the GitHub repository of the SLIPO project, along with a configuration file for the harvester to present its functionality. Although there is no installation procedure, some requirements have to be met. These are mainly the programming languages used since the scripts are not compiled into executables. Therefore, one has to install PHP in order to run PDH and python for PDE. The installation procedure varies depending on the operating system. Version 7 of PHP is recommended and python version at least 3.5.

PDH does not depend on any additional libraries, with the exception of PHP tidy which is recommended. Tidy comes as an extension of PHP (the corresponding library for Windows is php_tidy.dll). All other libraries come as default with PHP installation. PDE, on the other hand, depends on some third-party tools. An easy way to install packages for python is through the package management system called pip. pip can be installed directly or through the anaconda software on Windows. With pip installed, the following command:

```
pip3 install <package_name>
```

can be issued from the command line (where <package_name> is meant to be replaced with the actual name of the package to be installed) in order to install a package for python 3. The packages required for data exploration are the following: pandas, numpy, re, collections, unicodedata and json.

4.2. Configuration

Both PDH and PDE come with a main executable script. In case of PDH, the corresponding script harvest takes just one argument, the path of the configuration file in JSON format. Details about this file are given in the next subsection. For the case of PDE, configuration is given directly during the call of the corresponding script explore, as it will be described in detail in the last subsection.

In all cases of harvesting, the configuration file should be in JSON format. In this file, one should determine the type of harvester to use, with possible values being one of web, list, api or ckan, the output file name and, optionally, the maximum number of simultaneous connections.

4.2.1. HTML Harvester Configuration

Below we give an example of the structure of the configuration file for PDH-HTML, and subsequently we examine each parameter individually. Before we proceed, it has to be clear that this configuration file is intended for HTML harvesting, as the first argument (type) implies.

Code snippet: Sample configuration for PDH-HTML

```
// Settings for web harvester
{
    "type": "web",
    // The requested url or path to local file to obtain the POI list (json)
    "url": "https://www.example.com/stores/action-get?type=restaurants",
    // The sequence of fields, separated by semicolon, under which the list is available
    "fields": "collection;stores",
    // The title of the link identifier
    "link_id": "slug",
    // The base url for the individual links for each POI
    "base_url": "https://www.example.com/restaurant",
    // The variable name of the pagination
    "page_id": "page",
    // The identifier, in case it exists, for the coordinates
    "coordinates_id": "coordinates",
    // The identifier for the name of the POI
    "name_id": "store_title",
    // The tags which will be searched inside the html
    "tags": {
            "style": "results-list-cuisine",
            "city": "::::Τηλέφωνο",
            "address": "::Τηλέφωνο",
            "phone_number": "Τηλέφωνο",
            "fax": "Fax:",
            "site": "Site",
            "facebook": "facebook",
```

```
            "price_range": "Τιμή",
            "opening_days": "Ανοιχτό",
            "facilities": "facilities",
            "rating": "RestaurantRating",
    },
    // The maximum number of simultaneous connections (default: 5)
    "max_connections": 5
}
```

List creation

First and most importantly, one has to provide the harvester with the URL of the Web site to be harvested. This is not necessarily the base URL of the site but rather the URL which provides the list with the hyperlinks of Web pages containing the desired data. In our example, this is the following:

```
https://www.example.com/stores/action-get?type=restaurants
```

This is supposed to return either an HTML page which contains the hyperlinks under discussion or a JSON response. For the sake of our discussion, we suppose that the hyperlinks correspond to pages about restaurants.

First, we take the case that under the above URL an API listens responding with a JSON upon our request. To be specific, suppose the following structure for the response:

Example: Sample response with list of restaurants

```
collection: {
        pagination: {...},
        stores: [
                0: {
                        slug: restaurant0,
                        store_title: Restaurant 0,
                        ...
                },
                1: {
                        slug: restaurant1,
                        store_title: Restaurant 1,
                        ...
```

```
                },
                ...
        ],
        ...
}
```

In this example, we also suppose that the URL of each restaurant is constructed if the value corresponding to the slug key is concatenated with a base URL. Therefore, what we would like to collect is exactly the values of these slugs. First, we have to define where the slug is located, that is under collection and subsequently under stores. This is the reason in configuration file given above, the value "collection;stores" have been given to "fields". The "link_id" is the actual link identifier key, which above has been defined as "slug". Each value of slug will be appended to the value of "base_url".

Let's suppose, now, that the HTML of the web page under this URL has, among other, the following content:

Example: Sample snippet of the HTML code of a Web page

```
<ul class="main-restaurants-list">
        <li class="main-restaurants-list-item">
                <a class="restaurants-item-link" href="/restaurant1"> Restaurant1 </a>
        </li>
        <li class="main-restaurants-list-item">
                <a class="restaurants-item-link" href="/restaurant2"> Restaurant2 </a>
        </li>
</ul>
<a itemprop="secondary-restaurants-item-link" href="/restaurant3"> Restaurant3 </a>
```

In this example, what we wish to collect is the href attribute of all <a> HTML tags that are referring to restaurants. Therefore, we have two identifiers for these tags, which in this case are represented by the class attribute, although in general they could be any characteristic item. These identifiers are provided with the fields key of the JSON configuration file:

```
"fields": "restaurants-item-link;secondary-restaurants-item-link"
```

However, it is not uncommon that several links have to be followed before we arrive on a page containing the actual URL links. In this case the "fields" parameter should be an array of values. For example, it could be:

Example: Configuration snippet

```
"fields": [
```

```
            "areas-list-item-link;areas-city-link",
            "url"
],
"link_id": "url"
```

With this configuration, the harvester first creates a list of links contained under the areas-list-item-link and areas-city-link attributes. Subsequently, it visits these links and creates another one with links contained under the url attribute. This last list is the final one, which is used for harvesting.

Another case is that a JSON file listing all the URLs is in our disposal. Then, there is no need to fetch the list. Instead, the JSON file can be directly supplied to the configuration file. One can provide the path to a JSON file instead of a URL in the url entry of harvest.conf. The structure of the JSON file can be described as usual defining the appropriate values for the fields link_id and base_url.

In all cases, it is quite common that the results are paginated. One should provide the page identifier page_id to cover this possibility. Once such a value is provided to the harvester, the results will be iterated over the values found in this attribute.

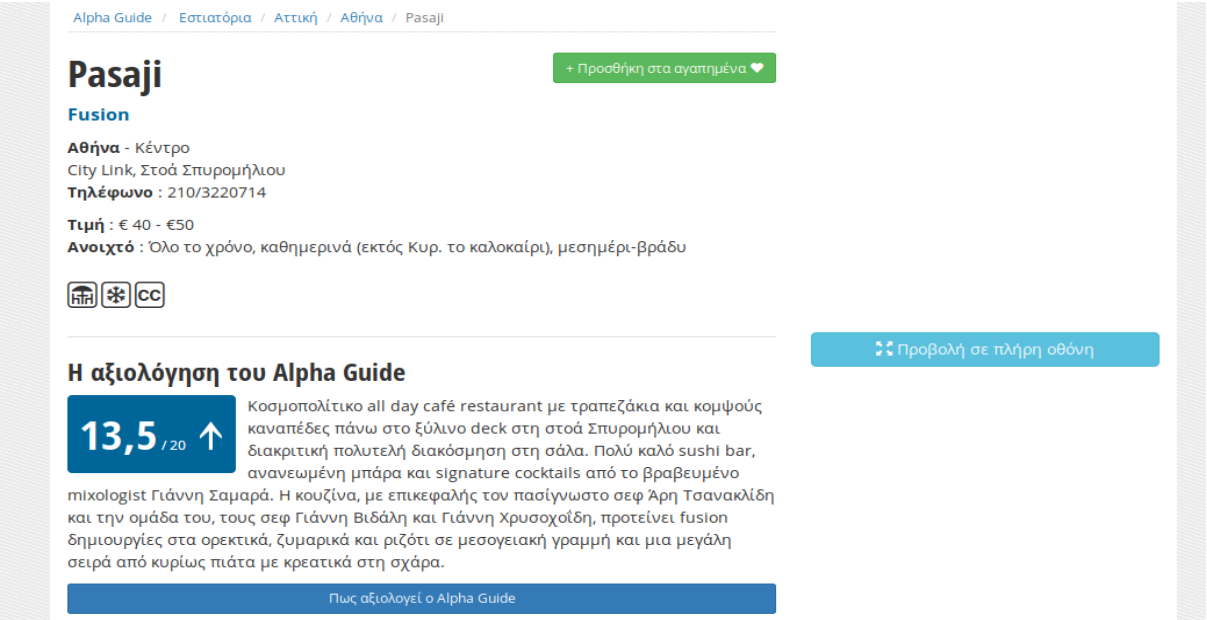
Finally, before we proceed to the configuration of the actual harvesting, it is important to note about two essential pieces of information that can be supplied inside the list. Namely, the title (name) of the POI and its coordinates. To cover this case, one can specify either the corresponding keys or the characteristic HTML attributes under the name_id and coordinates_id, accordingly, inside the configuration file.

Harvesting options

Once the final list with URLs has been constructed, the harvester starts searching for tags to populate the corresponding keys with values, either in a CSV file or in a JSON object in memory. The tags, as shown in the configuration example before, are defined in a key/value pair format. The key represents the name that the user gives to each field, i.e., the title of the column in a CSV file. The value is the element the harvester searches for inside the HTML structure.

As may have been noticed in the configuration example, some values are preceded with a semicolon (:). This semicolon has a special meaning. Each semicolon before a value is interpreted by the harvester as one place (one element) back. As will become clear in a while, during a presented example, this functionality is useful when there is no fixed point associated with the element which needs to be harvested. In this case, one has to find a fixed point and then it is possible to indicate its value and place by specifying the number of semicolons in front of it to be equal to the number of places before this point in the element under discussion.

We demonstrate this functionality by means of the following example. We assume that we would like to harvest a Web page about a restaurant with the following appearance:



The details in this Web page that we are interested in are the following:

1. Restaurant type (style): Fusion
2. City: Αθήνα
3. Address: City Link, Στοά Σπυρομήλιου
4. Phone number: 210/3220714
5. Price range: € 40 - €50
6. Opening days: Όλο το χρόνο, καθημερινά (εκτός Κυρ. το καλοκαίρι), μεσημέρι-βράδυ
7. Facilities: *the information contained in the three icons just after the opening days*
8. Rating: 13,5/20

For some of these details it is straightforward to find the appropriate rule. For example, for the phone number we could define a tag as

```
"phone_number": "Τηλέφωνο"
```

since beside the phone number there is the title "Τηλέφωνο". However, for most of the other pieces we have to dive into the HTML structure. Therefore, it is time to present the HTML code of this example page, at least the part involved in the details above:

```
686 <div class="details">
687 <p>
688 <span class="results-list-cuisine">Fusion</span>
689 </p>
690 <p>
691 <b>Αθήνα </b> - Κέντρο <br>City Link, Στοά Σπυρομήλιου <br>
692 <b>
693 Τηλέφωνο
694 </b>: 210/3220714<br />
695 <span style="display:none;">
696 <b>Fax: </b><br>
697 </span>
698 <span style="display:none;">
699 <b>Site</b>: <a href="http://" target="_blank"></a><br>
700 </span>
701 <span style="display:none;">
702 <span class="facebook-icon"></span><a href="" target="_blank">facebook</a>
703 </span>
704 </p>
705 <div class="clearfix"></div>
706 <p>
707 <span><b>
708 Τιμή
709 </b>: &euro; 40 - €50<br>
710 </span>
711 <span><b>
712 Ανοιχτό
713 </b>: Όλο το χρόνο, καθημερινά (εκτός Κυρ. το καλοκαίρι), μεσημέρι-βράδυ</span>
714 </p>
720 <ul class="facilities">
721 <li>
722 <div data-icon="B" class="icon" data-toggle="tooltip" data-placement="top" title="Σερβίρει και έξω"><span>Σερβίρει και έξω </span></div>
723 </li>
724 <li>
725 <div data-icon="b" class="icon" data-toggle="tooltip" data-placement="top" title="Κλιματισμός"><span>Κλιματισμός </span></div>
726 </li>
727 <li>
728 <div data-icon="q" class="icon" data-toggle="tooltip" data-placement="top" title="Δέχεται πιστωτικές κάρτες"><span>Δέχεται πιστωτικές κάρτες </span></div>
729 </li>
730 </ul>
731 <div class="clearfix"></div>
732 </div>
733 </div>
734 <ul class="list-inline specs" style="display:none;">
735 <li id="showReviews"><a href="#" id="show-reviews-detail">0
736 Αξιολογήσεις
737 </a></li>
748 <h3>
749 Η αξιολόγηση του Alpha Guide
750 </h3>
751 <span class="RestaurantRating">13,5<small> / 20</small>
752 <span class='restaurant-arrowup arrow-big'></span>
753 </span>
```

We can now observe the following:

1. Fusion is surrounded by a span tag with a unique class, therefore the first tag definition would be: "style": "results-list-cuisine".

2. The city name Αθήνα is not so trivial to harvest. It is not inside a unique tag neither has an identifier in front. Thus, we have to find a fixed point and define the position of Αθήνα relative to this. The fixed point we will pick in this example will be the element Τηλέφωνο. Αθήνα is then the fourth element before Τηλέφωνο. Therefore, the appropriate setting would be: "city": "::::Τηλέφωνο".

3. Using the same rationale, the setting for the address would be: "address": "::Τηλέφωνο".

In this manner, we can continue constructing the rest of the settings, resulting in the tags settings presented in the configuration example in the beginning of this section.

4.2.2. API Harvester Configuration

For PDH-API, the configuration file has a much simpler structure. First of all, the type attribute should be provided, and, in this case, it should be equal to api. The second essential option is the base_url parameter, which should correspond to the URL where the API is listening. As in every case, the output filename should also be provided. An optional attribute is max_connections used directly by cURL, which defaults to 5.

There are five additional configuration parameters:

- base_url : corresponds to the URL providing the list of the keys (ids).
- identifier : the name that the specific API uses for this key.
- query : this is optional and in case it is not provided all datasets are returned.
- other_params : also optional, this specifies additional parameters that could be attached to the request to the API.
- requests_limit : the maximum number of requests per minute.

4.2.3. CKAN Harvester Configuration

As in the previous cases, the configuration is provided in a JSON file. There are five parameters involved, with only two of them being mandatory. These are the parameters type and url. The former is set to ckan, whereas the latter is the URL of the data catalogue to be harvested. Additionally, one could determine the API version number, which by default is set to 3.

If no additional parameters are set, besides the two aforementioned ones, the CKAN harvester will attempt to fetch the entire contents of the catalogue. It will first retrieve the list with the total packages ids and subsequently it will retrieve the content of each package, merging all information in one file.

Alternatively, the user can choose to fetch only those datasets satisfying a specific query. In this case, it is possible to use one or both of the options query and filter_query. The former corresponds to a free text search, while with the latter can specify filters for searching in certain attributes. Boolean operators can also be used inside queries, as well as parentheses to define logical priorities. In case of filter queries, the values should be given in one of the two following formats:

```
"filter_query": "tags:<tag_value_to_search_for>"
```

or

```
"filter_query": "<vocabulary_name>:<tag_value_to_search_for>"
```

Concluding, if we need to harvest the entire data catalogue of, e.g., ckan.example.com, we should provide the harvester with the following configuration file:

Code snippet: Sample configuration for PDH-CKAN (without filtering)

```
// Settings for ckan harvester
{
    // The harvester type
    "type": "ckan",
    // The URL of the catalogue
    "url": "https://ckan.example.com",
    // Optionally, the CKAN API version
    "api_version": "3"
```

```
}
```

On the other hand, if we need to fetch only specific content, we could provide a configuration of the following type:

Code snippet: Sample configuration for PDH-CKAN (with filtering)

```
{
    // The harvester type
    "type": "ckan",
    // The URL of the catalogue
    "url": "https://ckan.example.com",
    // Optionally, the CKAN API version
    "api_version": "3",
    // Free text query
    "query": "This text should be included somewhere inside the dataset",
    // Filters
    "filter_query": "tags:(example1 OR (example2 AND example3))"
}
```

4.2.4. Configuration for POI Data Exploration

The PDE tool has a simpler configuration compared to PDH. In this case, we provide the necessary settings directly with the tool execution in the command line rather than defining them in a separate file. Each parameter has the following form:

```
parameter_name=parameter_value
```

There are four parameters, of which only one is mandatory. This is the filename, which specifies the full path of the file containing the dataset to be assessed. If only filename is supplied, then the tool returns a JSON file with the general statistics of the dataset.

In order to compute more detailed statistics for a specific column, the name of the column has to be supplied in the parameter column. In that case, the CSV file of the dataset has to include a first line with the column names. The value passed to column has to coincide with one of these names. If no other option is passed to the script, then a generic analysis is performed to the data contained inside this column. The results are returned in a JSON structure suitable for generating, by default, a pie chart. This can be altered by specifying the value of the parameter chart_type which can be either pie or bar.

The last parameter is category. This parameter can take any of the values generic, categorical, schedule, name, cost, address, phone or rating, with generic being the default. The corresponding functionality associated with each value of this parameter has been explained in Section 3.2.2.

Concluding, the syntax of the command issued to run the script with the available parameters is:

```
python  explore.py  filename=<filename>.csv  [column=<column_name>  [category=generic  |
categorical | schedule | name | cost |address | phone | rating] [chart_type=pie|bar] ]
```

4.3. Execution

In the following, we provide some examples demonstrating the execution of PDH and PDE. Through these examples, we explore the capabilities of the tools, and provide an explanation of the results.

4.3.1. Executing PDH

We have tested the harvesting toolkit in a variety of cases. To demonstrate its functionality, we present the following concrete examples:

- an execution of PDH-HTML on a Web site about restaurants located in Greece
- an execution of PDH-API on the Web site https://openchargemap.org
- an execution of PDH-CKAN on the European open data portal http://data.europa.eu.

For the first case, we provide an indicative example for harvesting POI data from the entertainment portal http://www.alpha-guide.gr. In this case, we have already created the list of URLs and saved it in JSON format. Thus, we use the following configuration file:

Example configuration file for the execution of PDH-HTML
{
// Using web harvester
"type": "web",
// Output path and filename
"filename": "data/alpha.csv",
// The requested url or path to local file to obtain the POI list (json)
"url": "./alpha.json",
// The sequence of fields, separated by semicolon, under which the list is available
"fields": "jsrestaurants",
// The base url for the individual links for each POI
"base_url": "http://www.alpha-guide.gr/el/restaurants",
// The title of the link identifier

```
    "link_id": 1,
    // The identifier for the POI's name
    "name_id": 0,
    // The tags which will be searched inside the html
    "tags": {
            "Type": "results-list-cuisine",
            "RestaurantRating": "RestaurantRating",
            "City": ":::Τηλέφωνο",
            "Address": "::Τηλέφωνο",
            "Phone_number": "Τηλέφωνο",
            "Fax": "Fax:",
            "Site": "Site",
            "Facebook": "facebook",
            "Price": "Τιμή",
            "OpeningHours": "Ανοιχτό",
            "Facilities": "facilities"
    },
    // The maximum number of simultaneous connections (default: 5)
    "max_connections": 5
}
```

We proceed to the second example of retrieving the information about the charging stations worldwide. The information about this kind of POIs is provided by openchargemap.org via an API. It is possible to retrieve all the available information if we iterate over the total set of country codes. Although the API does not explicitly provide a list with the available codes, it uses the 2-digit ISO code format which can be obtained from various sources. Below is the configuration file we use with which we obtained a dataset with over 70,000 entries (around 500Mb).

Example configuration file for the execution of PDH-API

```
// Settings for Api Harvester
{
    "type": "api",
    "base_url": "https://api.openchargemap.io/v2/poi/",
```

```
    "identifier": "countrycode",
    "list_url": "http://country.io/names.json",
    "filename": "data/ocm.json",
    "other_params": "maxresults=100000",
    "requests_limit": 2
}
```

Note that using a similar configuration, we have also obtained information about weather conditions from openweathermap.org.

Finally, we present an example execution of PDH-CKAN, demonstrating also the benefits offered by the SOLR search engine to filter which datasets to retrieve. Particularly, in this example we are going to harvest the European Union Open Data portal, using the configuration shown below.

Example configuration file for the execution of PDH-CKAN

```
{
    "type": "ckan",
    "filename": "data/ckan.json",
    "url": "https://data.europa.eu/euodp/data",
    "filter_query": "tags:('Science and technology' AND 'Environment')",
    "query": "title:'Agricultural'",
    "results": 10000,
    "max_connections": 5
}
```

In this specific example, we are interested only on those datasets tagged with both labels "Science and technology" and "Environment", and additionally only those containing the term "Agricultural" inside the title of the dataset. Using this configuration, PDH-CKAN fetches only the respective datasets. It turns out that there are only two datasets that satisfy these filters, which were retrieved in less than a second.

4.3.2. Executing PDE

Next, we present an example of the execution of PDE. We use a dataset obtained by harvesting a Web site providing information about restaurants in Greece. The dataset is provided in a CSV file (dataset.csv).

First, we obtain general information for this dataset by running:

```
python explore.py filename=dataset.csv
```

The output is written in a JSON file containing the general information about the dataset. A visualization of the results is shown in Figure 2. The table on the top left presents some basic information, including the name of the file, the number of rows and columns, as well as any possible errors during the execution of the script. The main information is provided by means of a dynamic tree diagram. The central node in this diagram represents the entire dataset. The children of this node are the fields the dataset contains (the titles of the columns in the corresponding CSV file). If a field contains distinct values, the number of which does not exceed a given threshold, then the field is considered categorical. Then, these distinct values of this field are presented in the diagram as child nodes of the respective node. In this example, we can notice that this is for instance the case for attributes style and payment. In this way, the user can quickly obtain a high-level overview of the contents of the dataset.

Next, to drill down to more detailed information about a specific column, the following command can be issued:

```
python explore.py column=best_for
```

In this case, a generic analysis is performed for the column entitled best_for. The resulting JSON response is included in the table below, including, for instance, the total number of values, the number of distinct values, the most frequent value, etc.

A more elaborate analysis can be performed in this particular case since this field is found to contain categorical data. Instead of the previous command, we can now run:

```
python explore.py filename=dataset.csv column=best_for category=categorical
```

Figure 3 shows a pie chart visualization of the more detailed results produced now.

Alternatively, we can choose to present these results in the form of a bar chart. In that case, we execute the following commnad:

```
python explore.py filename=dataset.csv column=best_for category=categorical chart_type=bar
```

which produces the bar chart depicted in Figure 4.

filename	dataset.csv
rows	1043
columns	11
errors	

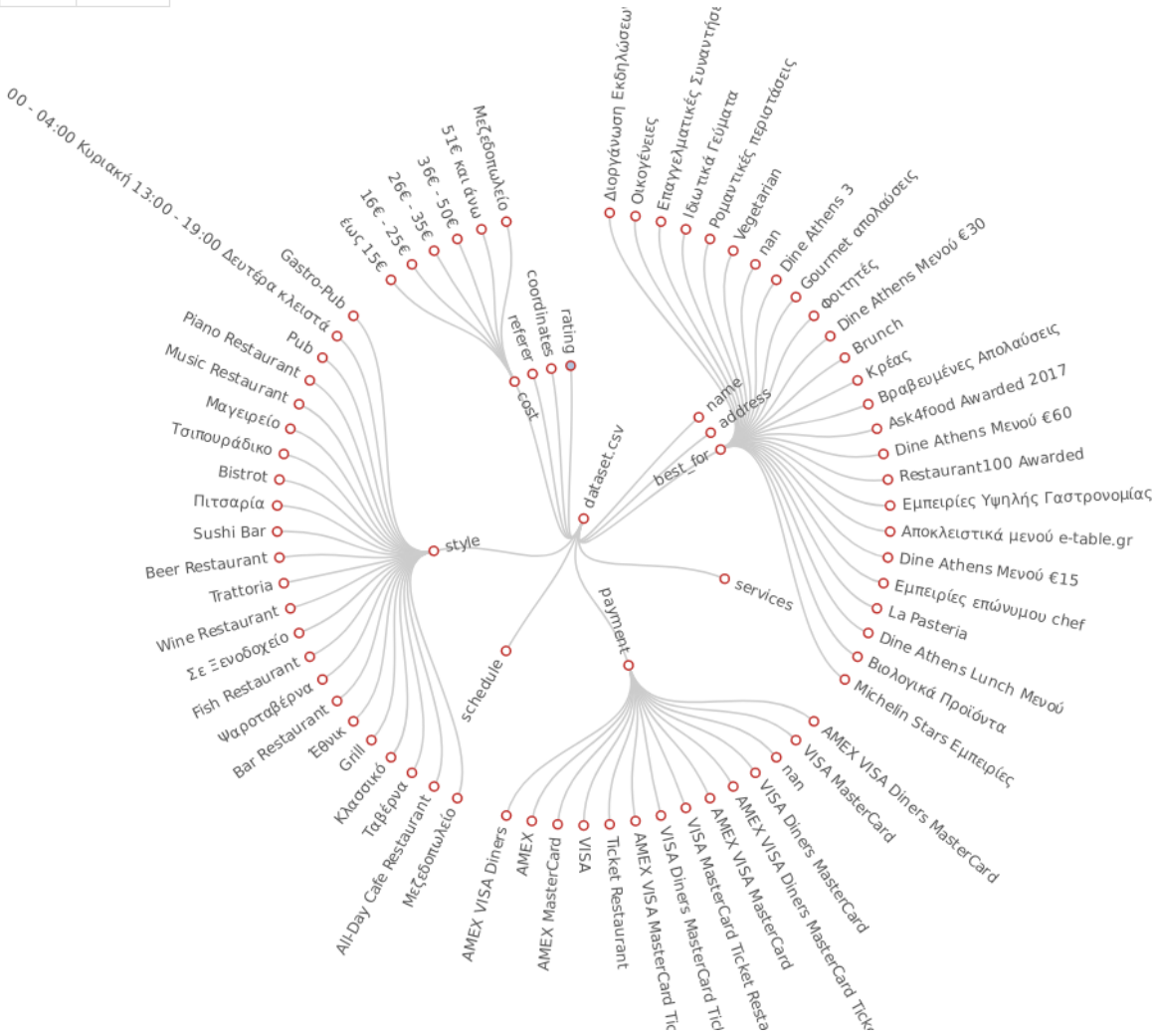


Figure 2: Example visualization of the general analysis of a CSV POI dataset with PDE.

Sample result from the analysis of a given attribue.
{
"description": {
"count": 2611,
"unique": 24,
"top": "Διοργάνωση Εκδηλώσεων",
"freq": 571,

```
            "null": 0,
            "minimum length": 5,
            "maximum length": 29
  },
  "generic": {
            "type": "pie",
            "data": [
            {
            "value": 2093,
            "name": "greek word(s) + \nE.g.:Ρομαντικές περιστάσεις"
            },
            {
            "value": 518,
            "name": "english word(s) + \nE.g.:Vegetarian"
            }
            ]
  }
}
```

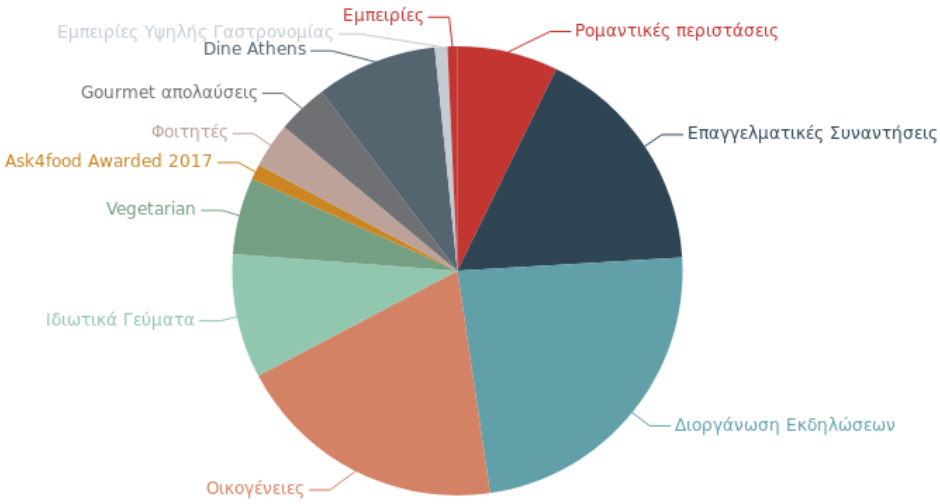


Figure 3: Example of a pie chart visualizing the analysis results of a given attribute.

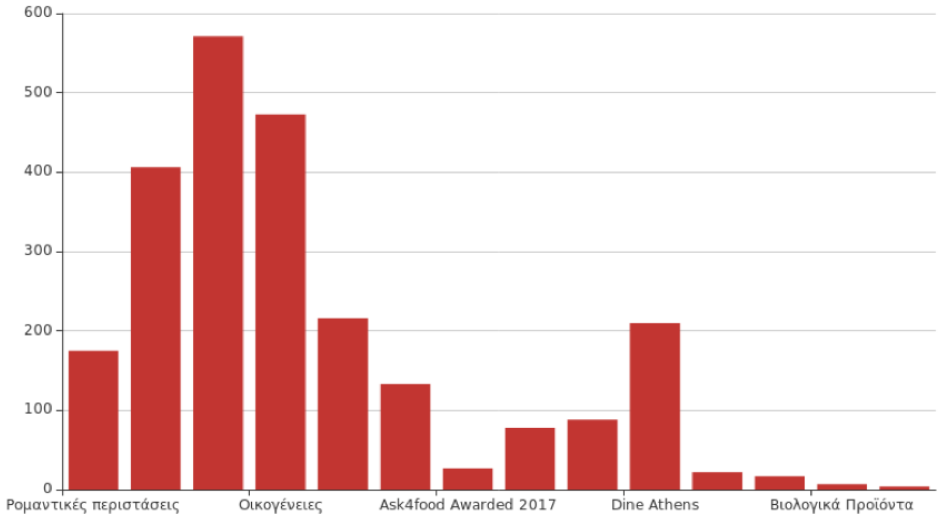


Figure 4: Example of a bar chart visualizing the analysis results of a given attribute.

Finally, we present two more cases. The first refers to a pie chart obtained by analysing the column corresponding to the name of the POIs (see Figure 5). In this case, information about the number of words and the language is given. The second involves the attribute address and is presented in Figure 6. In this case, one could retrieve information about the format of the address. In this specific example, we observe that none of the addresses contain a postal code. Most of them are in the form "words + number + words",

which likely indicates street name, street number and location. There exist also a few cases with two street numbers, which likely indicates addresses located in street corners.

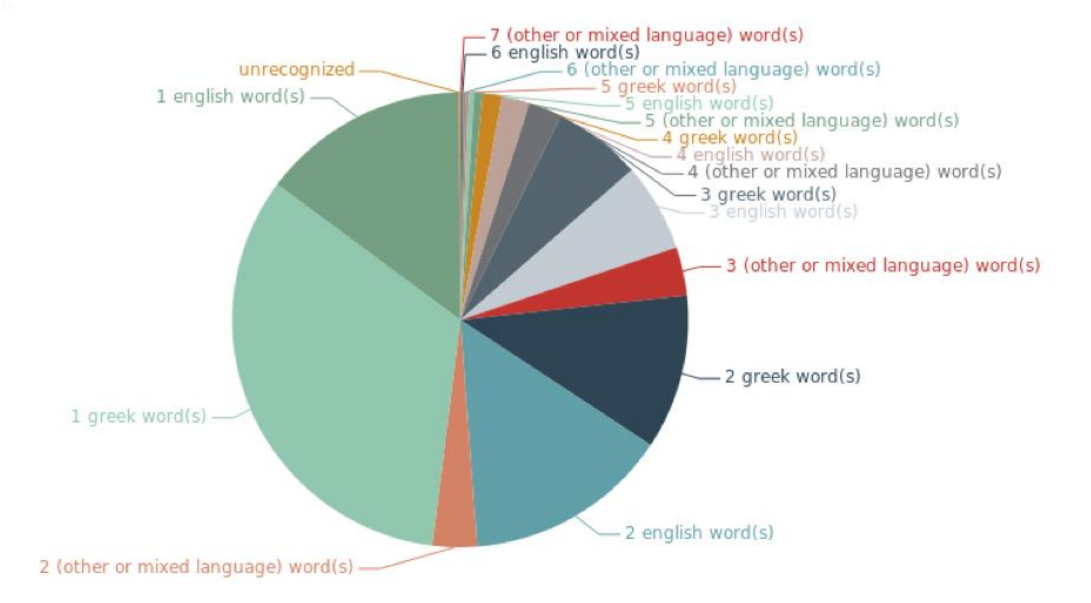


Figure 5: Example pie chart showing the result of the analysis on POI names.

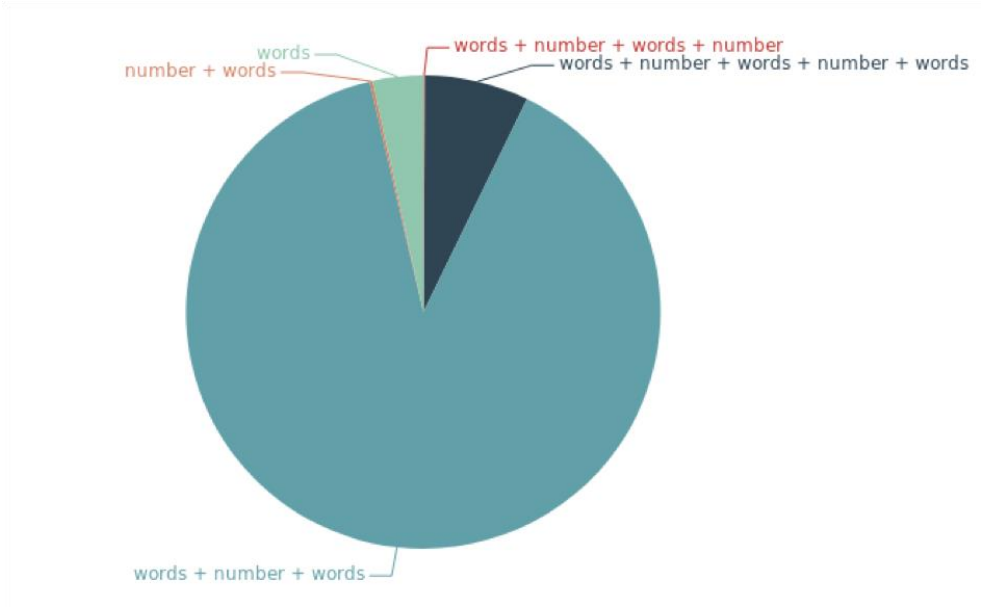


Figure 6: Example pie chart showing the result of the analysis on POI addresses.

References

[CKAN] The CKAN data portal platform. Available online at: https://ckan.org/

[DFMZ18] R. Domingues, M. Filippone, P. Michiardi, J. Zouaoui. A comparative evaluation of outlier detection algorithms: Experiments and analyses. Pattern Recognition 74: 406-421 (2018).

[DLS14] T. Dasu, J. M. Loh, D. Srivastava. Empirical glitch explanations. KDD 2014: 572-581.

[ODM] OpenDataMonitor. Available online at: https://opendatamonitor.eu/

[XPATH] XML Path Language. Available online at: https://www.w3.org/TR/1999/REC-xpath-19991116/