# D3.2 – EW-Shopp components as a service: transformation, linking and analytics

<table>
<thead>
<tr>
<th><strong>Deliverable n:</strong></th>
<th>3.2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Date:</strong></td>
<td>29 June 2018</td>
</tr>
<tr>
<td><strong>Status:</strong></td>
<td>Final</td>
</tr>
<tr>
<td><strong>Version:</strong></td>
<td>1.0</td>
</tr>
<tr>
<td><strong>Authors:</strong></td>
<td>Aljaž Košmerlj (JSI), Nikolay Nikolov (SINTEF), Michele Ciavotta (UNIMIB)</td>
</tr>
<tr>
<td><strong>Contributors:</strong></td>
<td>Patrik Zajec (JSI), Žan Palčič (JSI)</td>
</tr>
<tr>
<td><strong>Reviewers:</strong></td>
<td>David Čreslovnik (CE), Matej Žvan (CDE)</td>
</tr>
<tr>
<td><strong>Distribution:</strong></td>
<td>Public</td>
</tr>
</tbody>
</table>
# History of Changes

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Description</th>
<th>Revised by</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>24/4/2018</td>
<td>Definition of the Table of Contents</td>
<td></td>
</tr>
<tr>
<td>0.9</td>
<td>25/6/2018</td>
<td>Merged document ready for revision</td>
<td></td>
</tr>
<tr>
<td>0.9.1</td>
<td>28/6/2018</td>
<td>David’s comments fixed + minor updates</td>
<td></td>
</tr>
<tr>
<td>0.9.2</td>
<td>28/6/2018</td>
<td>Matej’s comments implemented</td>
<td></td>
</tr>
<tr>
<td>1.0</td>
<td>29/6/2018</td>
<td>Minor updates and typo checking</td>
<td></td>
</tr>
</tbody>
</table>
Executive Summary

This deliverable describes the current implementation of the transformation, linking and analytics services in the EW-Shopp platform. Each set of services is described with an emphasis on their set up and use. The services’ APIs for deployment and operations are specified and examples are provided.

The work describes in this document fulfils the requirements and follows the technical specifications described in deliverables from work package 1. The services are parts of the components in the overall platform architecture developed within work package 2 and they support the business cases in work package 4.
# Table of contents

History of Changes .................................................................................................................. 2
Executive Summary ...................................................................................................................... 3
List of figures .............................................................................................................................. 6
List of tables ............................................................................................................................... 7
Acronyms ..................................................................................................................................... 8

Chapter 1  Introduction .............................................................................................................. 10
  1.1  Objectives and Scope ........................................................................................................ 10
  1.2  Relationship to Other Deliverables ................................................................................ 10

Chapter 2  Transformation Services ......................................................................................... 11
  2.1  Introduction and Terminology ........................................................................................ 11
  2.2  Data Wrangler component .............................................................................................. 12
    2.2.1  Data cleaning and data enrichment ........................................................................... 12
    2.2.2  Consuming data ......................................................................................................... 13
  2.3  Processing component ..................................................................................................... 13
    2.3.1  Data ingestion and workflow processing .................................................................... 14
    2.3.2  Data workflow steps library .................................................................................... 14

Chapter 3  Linking Services ...................................................................................................... 16
  3.1  Inter-linking ..................................................................................................................... 18
  3.2  Example: Geospatial Data Reconciliation and Extension ................................................ 19
  3.3  Reconciliation in EW-Shopp .............................................................................................. 21
    3.3.1  Architecture .............................................................................................................. 21
    3.3.2  Workflow .................................................................................................................. 24
  3.4  Reconciliation and Extension Services ............................................................................. 26
    3.4.1  Wikifier ....................................................................................................................... 26
    3.4.2  GeoNames ................................................................................................................. 27
    3.4.3  Google Geotargets ................................................................................................. 27
    3.4.4  Google ProductsServices ...................................................................................... 28
    3.4.5  GFK Products .......................................................................................................... 29
    3.4.6  OpenRefine Reconciliation Service API .................................................................. 30
  3.5  Reconciliation Services development .............................................................................. 34
List of figures

**Figure 1.** Transformation services as part of the first implementation of the EW-Shopp platform ........................................11

**Figure 2.** File system folder assignment in the Processing component .................................................................14

**Figure 3.** An excerpt of the JOT Business case: an example of reconciliation ..........................................................17

**Figure 4.** An example of data extension. ..........................................................................................................................18

**Figure 5.** Interlinking among datasets ..........................................................................................................................19

**Figure 6.** Interlinking services architecture ...................................................................................................................22

**Figure 7.** The interlinking service architecture. ................................................................................................................23

**Figure 8.** The data workflow to generate a transformation and enrichment pipeline. ......................................................25

**Figure 9.** The data workflow to apply the transformation model to the corporate datasets. ...........................................26

**Figure 10.** Overview of QMiner architecture ...................................................................................................................36

**Figure 11.** Data analysis workflow in QMiner ..................................................................................................................37
List of tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Short references for project partners</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>Links among datasets in EW-Shopp</td>
<td>21</td>
</tr>
<tr>
<td>3</td>
<td>An example of collected IDs by the reconciliation service</td>
<td>24</td>
</tr>
<tr>
<td>4</td>
<td>Overview of the Geotargets dataset</td>
<td>27</td>
</tr>
</tbody>
</table>
### Acronyms

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>BC</td>
<td>Business Case</td>
</tr>
<tr>
<td>CSV</td>
<td>Comma-Separated Values</td>
</tr>
<tr>
<td>EAN</td>
<td>International Article Number (formerly European Article Number)</td>
</tr>
<tr>
<td>GPL</td>
<td>GNU General Public License</td>
</tr>
<tr>
<td>GRS</td>
<td>GeoNames Reconciliation Service</td>
</tr>
<tr>
<td>GSW</td>
<td>GeoNames Search WebService</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext Transfer Protocol</td>
</tr>
<tr>
<td>JAR</td>
<td>Java ARchive file</td>
</tr>
<tr>
<td>JSON</td>
<td>JavaScript Object Notation</td>
</tr>
<tr>
<td>PRS</td>
<td>Product Reconciliation Service</td>
</tr>
<tr>
<td>RDF</td>
<td>Resource Description Framework</td>
</tr>
<tr>
<td>REST</td>
<td>Representational State Transfer</td>
</tr>
<tr>
<td>SKU</td>
<td>Stock Keeping Unit</td>
</tr>
<tr>
<td>SPARQL</td>
<td>SPARQL Protocol and RDF Query Language</td>
</tr>
<tr>
<td>SSL</td>
<td>Secure Sockets Layer</td>
</tr>
<tr>
<td>SVM</td>
<td>Support Vector Machine</td>
</tr>
<tr>
<td>TSV</td>
<td>Tab-Separated Values</td>
</tr>
<tr>
<td>UI</td>
<td>User Interface</td>
</tr>
<tr>
<td>VPN</td>
<td>Virtual Private Network</td>
</tr>
<tr>
<td>WRS</td>
<td>Wikifier Reconciliation Service</td>
</tr>
</tbody>
</table>

Short references may be used to refer to project beneficiaries, also frequently referred to as partners. References are listed in Table 1.
### Table 1. Short references for project partners

<table>
<thead>
<tr>
<th>No.</th>
<th>Beneficiary (partner) name as in [GA]</th>
<th>Short reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>UNIVERSITÀ DEGLI STUDI DI MILANO-BICOCCA</td>
<td>UNIMIB</td>
</tr>
<tr>
<td>2</td>
<td>CENEJE DRUZBA ZA TRGOVINO IN POSLOVNO SVETOVARJE DOO</td>
<td>CE</td>
</tr>
<tr>
<td>3</td>
<td>BROWSETEL (UK) LIMITED</td>
<td>BT</td>
</tr>
<tr>
<td>4</td>
<td>GFK EURISCO SRL</td>
<td>GFK</td>
</tr>
<tr>
<td>5</td>
<td>BIG BANG, TRGOVINA IN STORITVE, DOO</td>
<td>BB</td>
</tr>
<tr>
<td>6</td>
<td>MEASURENCE LIMITED</td>
<td>ME</td>
</tr>
<tr>
<td>7</td>
<td>JOT INTERNET MEDIA ESPAÑA SL</td>
<td>JOT</td>
</tr>
<tr>
<td>8</td>
<td>ENGINEERING – INGEGNERIA INFORMATICA SPA</td>
<td>ENG</td>
</tr>
<tr>
<td>9</td>
<td>STIFTELSEN SINTEF</td>
<td>SINTEF</td>
</tr>
<tr>
<td>10</td>
<td>INSTITUT JOZEF STEFAN</td>
<td>JSI</td>
</tr>
</tbody>
</table>
Chapter 1  Introduction

The EW-Shopp project is building a platform, which supports businesses in processing their data, connecting them to external data sources and analysing them to produce insights and operational models to inform and power their business processes. Each step of this effort is supported by a set of tools developed by project partners to perform the individual tasks. These tools are wrapped in services to simplify integration into existing systems, support modular development and enable simple interchange of individual steps. This way businesses can efficiently tailor the EW-Shopp environment to their needs. This document describes how these services are set up, deployed and used.

1.1 Objectives and Scope

This deliverable describes the current implementation of the transformation, linking and analytics services in the EW-Shopp platform. It collects work done mainly by three technical partners, SINTEF, UNIMIB and JSI, each responsible for one stage in the EW-Shopp platform. The overall architecture, that these services are part of, has already been specified and described in other documentation (see deliverable D2.2 [1]). Here the focus is on description of functionality of individual services and their setup and operational APIs.

The document is organized as follows. The rest of the introduction in this chapter, Chapter 1, lists the relationships to other project deliverables. Chapter 2, Chapter 3 and Chapter 4 present the transformation, linking and analytics services respectively. Each chapter describes the theoretical and technical details needed for setting up and running their services. Finally, Chapter 5 concludes the document with a short overview and a note on future development.

1.2 Relationship to Other Deliverables

The contents of this document are strongly related to several other deliverables, both already finished as well as upcoming. The specification of the overall platform architecture, which connects these services, was described in deliverable D2.2 [1]. The services support the project pilots described in deliverable D4.2 [2] and follow the specifications and requirements from deliverables D1.1 [3], D1.2 [4], D1.3 [5] and D3.1 [6]. Outputs and results of these services will be used in the visualization and navigation services described in the upcoming deliverable D3.3 and any future updates to them will be described in the final platform description in deliverable D2.4.
Chapter 2  Transformation Services

This chapter discusses the transformation services of the EW-Shopp platform, which provides functionalities for scalable clean-up, transformation and enrichment (using the Linking Services) of data.

2.1 Introduction and Terminology

This chapter describes the current implementation of the transformation services with a focus on the extensions for the EW-Shopp platform. It also discusses the ways of interacting with the respective platform components to take advantage of the platform services. Figure 1 shows the scope of the Transformation services.

The Transformation services include the Data Wrangler component, Processing component and the various interfaces for connecting them together in the EW-Shopp platform. The Data Wrangler component is implemented by a combination of services – DataGraft\(^1\) and Grafterizer\(^2\) for data manipulation and asset management and ASIA/ABSTAT for enrichment (see Chapter 3 ). The chapter

\(^1\) https://datagraft.io/
\(^2\) https://grafterizer.datagraft.io/
includes information about the APIs for integration with the core data layer and for consuming sample data for producing the scalable templates for data cleaning, preparation and transformation, which are consumed by the Processing component. Details on the implementation of the Data Wrangler component can be found in deliverable D2.2 [1].

The Processing component implements a data workflow engine, which can execute steps, chosen from a library of pre-defined workflow step templates. The data workflows of the Processing component will be declared and coordinated by the System Orchestration component (see [1]).

Finally, this section discusses the interfaces between the Processing component and the Data Storage layer, which deals with delivering data for large-scale transformations.

## 2.2 Data Wrangler component

The Data Wrangler is a composite component built upon the DataGraft platform and extended with semantic enrichment and Big Data processing capabilities. DataGraft provides tools for cleaning and transformation of tabular data into RDF\(^3\) and graph generation mappings (more details can be found in [1]). DataGraft comes with an interactive web application, named Grafterizer, which serves as user interface helping platform users to transform data from a tabular format into a graph format. The transformation supports both cleaning and graph mapping steps. Transformation steps on rows (add, drop, filter, duplicate detection etc.), columns (add, drop, rename, merge etc.) and entire data set (sort, aggregate etc.) are provided together with visualization of the result after each step. The mapping to ontologies or vocabularies is performed on the cleaned-up data.

### 2.2.1 Data cleaning and data enrichment

Grafterizer operates based on atomic steps and implements the pipe-and-filter, whereby each operation is performed on one line of input and then sent further in the pipeline. Pipeline steps are implemented in Clojure\(^4\) and the UI serves as a Clojure code generator. Then, the UI calls a back-end service, called Graftwerk in order to retrieve results from transformation steps in the pipeline. Apart from the client-server mode of operation, Grafterizer supports composition of executable Java Archives (JARs), which encapsulate the transformation logic and all the necessary dependencies to run it (general-purpose/transformation specific library functions, and other Java dependencies for running Clojure code).

In EW-Shopp, in order to support scalable data enrichment, Grafterizer has been integrated with two services – ASIA and ABSTAT. Thereby, Grafterizer is currently being extended to support more complex reconciliation services, which are described in Chapter 3. The extensions will allow users of Grafterizer to reconcile data through extensions in the pipeline APIs through calls to the ASIA web service. To support this, we are currently developing code to integrate the Graftwerk back-end, JAR generation service, and processing component. Graftwerk will be extended to execute REST calls to the ASIA service to retrieve reconciled data given an input and will cache intermediary results in a

\(^3\) [https://www.w3.org/RDF/](https://www.w3.org/RDF/)

\(^4\) [https://clojure.org/](https://clojure.org/)
rotary cache so that they can be quickly retrieved at runtime if multiple calls for the same transformation are sent. The JAR generation template will be extended with a local variable that will store temporary results while the transformation is being performed over the input. Finally, the Processing component will run a cache component (e.g., an in-memory database), which will be made available across the entire workflow as a service. Thereby, any clean-up/enrichment steps will be able to access previously retrieved enrichment data and use it without direct calls to the ASIA service.

### 2.2.2 Consuming data

The Data Wrangler provides functionalities to interactively define and execute cleaning, pre-processing and transformation over data. The inputs to the data transformations are delivered in two ways – through the user interface of Grafterizer, and as parameters to the self-contained executable JARs that get produced based on the data transformation templates.

For small-scale data (up to a few megabytes of data), the web-based user interface of Grafterizer provides the possibility of directly loading the data. This mode of operation provides the possibility to process sample data from the EW-Shopp platform users. The transformations of the sample data can then be scaled up for very large volumes by the System Orchestration/Processing components. The inputs to the user interface can be uploaded directly to the Grafterizer user interface when a user creates a new transformation or uses one of the wizards for data publication on the DataGraft platform. Alternatively, since files are also managed assets on the DataGraft platform, users can choose to upload their files and register them in the DataGraft catalogue. Then, these files can be loaded in the Grafterizer user interface from a list of user assets in the wizard, or by directly choosing them inside of the Grafterizer interface.

For larger-scale transformations (a few hundred megabytes to a gigabyte), the Data Wrangler component provides the possibility to use executable JARs based on template data transformations defined in the Grafterizer user interface. In this mode, the input files are provided as a parameter when invoking the "java -jar" command (e.g., `java -jar inputFile.csv outputFile.csv`; results in this case will be stored in `outputFile.csv`). For larger volumes, the executable JAR files are used by the Processing component for horizontally scaling the data transformations (see sub-section 2.3).

### 2.3 Processing component

The Processing component of the EW-Shopp platform is responsible for the execution of data wrangling at scale as well as pre- and post-processing operations. The data workflows, which implement this process, are composed of steps that are encapsulated within Docker\(^5\) containers and exposed as a service so that new data can be processed continuously. The Docker containers are controlled by a container orchestration system – in the current implementation – Rancher\(^6\). More details on the implementation of the Processing component can be found in deliverable D2.2 [1].

---

\(^5\) [https://www.docker.com/](https://www.docker.com/)
\(^6\) [https://rancher.com/](https://rancher.com/)
2.3.1 Data ingestion and workflow processing

Interfacing of workflow steps is done with the help of the shared filesystem of the EW-Shopp platform. The shared filesystem stores the input to the data workflows as well as all intermediate results of workflow steps. Each workflow step is organised and treated by the Processing component as an independent filter. Steps are assigned an input folder, work folder, output folder in the file system as well as any other parameters that have to do with the execution of the workflow and other user preferences. Each step is instantiated from a Docker image, which is built in advance and made available from a workflow step library. Figure 2 shows the folder assignment scheme in the shared file system.

![Diagram showing folder assignment scheme](image)

In order to share intermediate results between steps, the input and output folders of each steps are chained, whereby the output folder of a step is set as the input folder for the next steps. The input, work and output folders are set up in advance and currently (in the first prototype, see [1]) manually pending the implementation of the System Orchestrator, which will have the task of setting up the folder structure and parameter assignment within a data workflow.

2.3.2 Data workflow steps library

As mentioned, the workflow steps library is made up of reusable Docker images that can be chained together to form a data workflow. Currently, we have implemented the following set of generic library steps:
• Decompress archive – decompresses all files from all archives in the input folder and puts the contents in the output folder

• Split files in chunks – splits all files in a folder in pre-determined chunks and stores the chunks in the output folder

• Convert TSV to CSV – converts all tab-separated values (TSV) files in the input folder to comma-separated values (CSV) files and stores the result in the output folder

• Transform using a Grafterizer data transformation template – invokes the executable JAR of a Grafterizer-based transformation and stores the result of the transformation in the output folder. This step takes as a parameter the executable JAR of the data transformation.

• Map CSV data to ArangoDB7 compatible JSON file format – maps CSV files in the input folder to the JSON format that is consumed by ArangoDB and stores the result in the output folder. This step takes the JSON representation (node and edge collections, see [1]) of the Grafterizer data transformation as input.

• Upload data to ArangoDB – uploads all edge and node collections from a folder to an ArangoDB instance. This step takes as input the address of the ArangoDB instance where the data should be imported, as well as the access credentials (if necessary).

Each workflow step is organised as a Docker image with a Bash entry script, which implements the high-level operations and calls the necessary application (e.g., GZip, Java runtime, NodeJS application, etc.) and takes care of moving the files from the input to the work folders of each instance. Furthermore, in order to achieve horizontal scalability of each step, it is important to enable multiple workflow step instances to process data in parallel, without concurrent data workflows steps interfering with each other. Therefore, each workflow step synchronises operations with other steps using low-level (atomic) file system operations to lock files that are currently being processed. In that way, no other service can attempt to obtain a file for processing that is already being processed by another instance of the workflow step image. Input, work and output folder locations for each step are mounted as volumes of the Docker images that implement the workflow step. Any additional parameters needed to implement the workflow step are passed as environment variable assignments. This ensures that the data workflow can be controlled on a high level (to be implemented in the second version of the EW-Shopp platform by the System Orchestration component) and can be composed into mixed workflows through a high-level user interface.

The currently implemented data workflow steps templates in the EW-Shopp platform are publicly available on GitHub8.

---

7 https://www.arangodb.com/
8 https://github.com/ew-shopp/ingestion_bc4_p01

EW-Shopp GA number: 732590 H2020-ICT-2016-2017/H2020-ICT-2016-1
Chapter 3 Linking Services

In EW-Shopp jargon the Inter-linking services represent a set of web services providing advanced and multi-domain Reconciliation and Extension support to both the definition of the data transformation pipeline (design time) and its execution (run time).

The term Reconciliation (or entity matching) refers to the set of functions of a system that associates a text string (and possibly a context) with a list of entities belonging to a knowledge base. These entities must correspond to the input string according to some criterion. The entities are identified by means of an identifier that has meaning in an identifier space (also called System of Identifiers).

Examples of shared identifier spaces are:

1. Wikidata ID
2. location identifiers provided by the Geonames dataset
3. SKU (Stock Keeping Unit) provided as identifiers for products.

Furthermore, we consider shared identifier spaces also syntax that supports the intentional specification of objects in infinite sets; for example, numbers, pairs of longitude and latitude, which identify specific points in a coordinate system.

Reconciliation services need to be able to address and resolve a number of issues effectively. Just to name a few, we can refer to the following problems:

1. Multi-lingual, i.e. the string can refer to one or more words in a language other than the one used in the knowledge base of reference;
2. mismatching between the string and the official name of the entity, which may be due to several factors such as abbreviations or typos;
3. polysemic words, i.e., those with different meanings and able to correspond to more than one entity with the same level of confidence.

Let’s consider the following example that aims at clarifying the reconciliation functionality. In the upper part of Figure 3, a table with data taken from JOT Business case is presented. Among other information, this table features two columns, Region and Category, which contain data related to geography and business categorizations, respectively. We executed an entity linking procedure on both columns against different knowledge bases to obtain GeoNames ID and DMOZ Categories. The result of the reconciliation is presented in the lower table in the figure. We have now two more columns (in green colour) featuring unique identifiers. It is worth noticing that disambiguation in the case of categories can be challenging; non-trivial techniques had to be applied to reconcile, for instance, the label “travelTurism” to “DMOZ/Recreation/Travel”.

EW-Shopp

GA number: 732590

H2020-ICT-2016-2017/H2020-ICT-2016-1

16
The term **Extension**, on the other hand, denotes a collection of functionalities, complementary and consequent to reconciliation, which enable the enrichment of the work data set with information collected in a reference knowledge base. To be more precise, once a property string has been disambiguated and linked to an identifier belonging to an entity in a specific space, it is possible to use that same identifier to extract information related to that entity from the knowledge base.

Thus, essentially, an extension service on a particular knowledge base has to:

1. Provide the caller with a list of metadata corresponding to the properties of the entity that can be extracted;
2. Given an ID and the specification of a property, extract and return the value of that property.

Continuing with the previous example, we can visually show how the extension process works (Figure 4). The aim is enriching the working data set with information about the number of events happened on a certain date, in a certain region, and with a reference to a specified category. The EventRegistry service is used as information source, the information about the position, category and date is taken from the columns **GeoNames Id**, **DMOZ**, and **Date**, respectively. The result is a data set extended with a new column (in light red colour) presented in the lower part of the figure.
It is important to note that reconciliation and extension are two closely dependent activities because it is not possible to proceed with the extension if the reconciliation process has not been correctly carried out.

### 3.1 Inter-linking

In the previous paragraph we briefly described the concepts of reconciliation and extension concerning a single space of identifiers. The choice to closely pair these functionalities is quite common and, as an example, is implemented in this way in OpenRefine. In the process of definition and execution of a pipeline of transformation and enrichment, however, the scenario in which the working data set must be extended using information available in different knowledge bases and accessible by means of identifiers belonging to different spaces is pretty common. One possibility would be to reconcile several times with different services in order to collect identifiers from the different knowledge bases. Although this route is feasible, it is clearly cumbersome, repetitive and error-prone due to the different precision with which the services are able to perform the entity matching process. The idea behind our approach consists, in layman’s terms, in being able to generate mappings among multiple spaces of identifiers and use them to translate with high fidelity one identified into another and use the latter to query its own knowledge base.

One of the objectives of the inter-linking service foreseen in EW-Shopp is to provide users with different single-domain reconciliation and extension services in a transparent way. Obviously, those
services whose identifiers refer to the same class can be used together. In EW-Shopp we have identified three classes of identifiers, namely spatial, products, and categories identifiers.

### 3.2 Example: Geospatial Data Reconciliation and Extension

In what follows, we present a short explanatory example concerning only spatial reconciliation and enriching.

In this example, the inter-linking services uses GeoNames as central and reference data set.

GeoNames has been selected because:

1. it is the largest dataset describing geo entities,
2. it is a multilingual dataset for entities labels, and
3. the administrative hierarchy for a given entity can be easily accessed (also via APIs).

The GeoNames RDF dump contains information for about 11.7 million geo entities (named Feature) as of March 2018. The other dataset to be considered is DBpedia-2016-10 as it describes 6.6 million entities, where 1.9 million have geo coordinates and 840k are places. Moreover, being DBpedia a cross-domain dataset, it contains various descriptive data for geospatial entities, thus it is often considered a very effective dataset for reconciliation. Another relevant dataset is Wikidata, which contains 3,496,401 triples (3,583,882 distinct instances) that have a link to GeoNames.

Among links to GeoNames, Wikidata has 29,783 links to Yahoo! WOEID, and 4,004 links to Facebook Places IDs. There exist also 1,355 direct links to NUTS codes. While in DBpedia instances are linked to GeoNames datasets by an owl: sameAs relation (links that declares two items to be identical). Except of owl:sameAs links, there are many links such as rdfs:seeAlso from DBpedia to local datasets (e.g. Embrune in France is linked through owl:sameAs to the database of INSEE, the statistical institute of France). The datasets considered in the platform are shown in Figure 5.

![Figure 5. Inter linking among datasets.](image-url)
Suppose a user is interested in enriching a business data set with information from different Knowledge Bases for geo-marketing. In the working dataset, only toponymy and country names are present in each row. Without the inter-linking service all the reference data sets should have been queried separately, that is she should have made a query of the form Milan, Italy (city, country) in each of the most common geospatial datasets such as GeoNames, DBpedia and Wikidata. In this situation, additional information and the geospatial ID should be retrieved from each dataset. Particularly, in this scenario:

1. In DBpedia she finds that there are about 45 properties describing the resource of Milan, Italy. There are two “sameAs” links from DBpedia connecting this resource to the ones in GeoNames. In this case she extracts two GeoNames ID for the resource of Milan.
2. Accessing such links, she could verify that one of them is wrong in our context as it refers to the city of Milan in Quebec, Canada. For her it is quite simple to decide which one to consider as she can verify the information contained in each link.
3. She makes the same query in GeoNames dataset and retrieves among other information also the information about administrative hierarchy and the alternate names in different languages.
4. The information collected so far is not enough as for her geo-marketing scooping she also needs to know identifiers such as NUTS, Facebook Places, WOEID, Google’s Code, etc. She can have some of these identifiers in Wikidata dataset. Thus, she searches Wikidata for Milan, and obtains a list of candidates for the query.
5. After understanding which the right resource is, she accesses the link containing the descriptive information required. Among other information there are also different identifiers that identify equivalent resources in different KBs, such as ISTAT ID, GeoNames ID, Facebook Places ID, etc.

Eventually, she can find all the required information, but such a process is time consuming and requires a huge effort, since it requires expertise in writing SPARQL queries and competences in different integration platforms.

The inter-linking service (as far as the spatial reconciliation is concerned) can be queried in three different ways:

1. by searching for a toponymy (e.g., Rome, Paris, etc.),
2. coordinates (e.g., 41.89193, 12.51133; 48.85341, 2.3488, etc.), or
3. by querying the whole address (e.g., via degli Olmetti 5B 00060 Formello (ROMA); 6, rue Arsene Houssaye, 75008, Paris).

However, a user may already have one identifier belonging to one knowledge base and needs to retrieve the other IDs. For this reason, the entry point can be any of the reference data set, which the user can select a priori or make a query without selecting the source data set.
Table 2 shows how the reconciliation process occurs among datasets, and for each of them we give details whether the link among datasets is direct or indirect (meaning that a dataset can be reached through other datasets). In the case a user searches a toponym, we make an exact match in GeoNames labels, and later retrieve all the other geospatial IDs from the platform. Reverse geocoding is used when coordinates are the information in input, while in the case of a complete address is queried, we parse the address and trace it back to the toponym case.

Table 2. Links among datasets in EW-Shopp.

<table>
<thead>
<tr>
<th>Name</th>
<th>GeoNames</th>
<th>Wikidata</th>
<th>DBpedia</th>
<th>Google Code</th>
<th>Toponymy</th>
<th>Coordinates</th>
<th>Complete Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>GeoNames</td>
<td>Direct</td>
<td>Indirect</td>
<td>Direct</td>
<td>Direct</td>
<td>Direct</td>
<td>Direct</td>
<td>Parsing</td>
</tr>
<tr>
<td>Wikidata</td>
<td>Direct</td>
<td>Direct</td>
<td>Indirect</td>
<td>Direct</td>
<td>Direct</td>
<td>Direct</td>
<td>Parsing</td>
</tr>
<tr>
<td>DBpedia</td>
<td>Indirect</td>
<td>Direct</td>
<td>Indirect</td>
<td>Direct</td>
<td>Direct</td>
<td>Direct</td>
<td>Parsing</td>
</tr>
<tr>
<td>Google Code</td>
<td>Direct</td>
<td>Indirect</td>
<td>Indirect</td>
<td>Direct</td>
<td>Reverse</td>
<td>Geocoding</td>
<td>Direct</td>
</tr>
<tr>
<td>Toponymy</td>
<td>Direct</td>
<td>Direct</td>
<td>Direct</td>
<td>Direct</td>
<td>Reverse</td>
<td>Geocoding</td>
<td>Parsing</td>
</tr>
<tr>
<td>Coordinates</td>
<td>Direct</td>
<td>Direct</td>
<td>Direct</td>
<td>Indirect</td>
<td>Reverse</td>
<td>Reverse</td>
<td>Geocoding</td>
</tr>
<tr>
<td>Complete Address</td>
<td>Parsing</td>
<td>Parsing</td>
<td>Parsing</td>
<td>Parsing</td>
<td>Parsing</td>
<td>Geocoding</td>
<td></td>
</tr>
</tbody>
</table>

3.3 Reconciliation in EW-Shopp

The aim of this section is to provide a general picture, from the architectural and workflow point of view, about how the inter-linking services (hereinafter we will use the plural form to refer to both the central inter-linking service and the underneath reconciliation/extension services) interact with the EW-Shopp platform both at design time and at runtime. In particular, the interactions and data flow with ASIA and the Big Data Runtime component are detailed and discussed.

3.3.1 Architecture

The main aim of the inter-linking services is to support users in an efficient and effective way in the reconciliation process. At the moment this document is written, the inter-linking services are composed of five reconciliation and extension services, each of which refers to a system of identifiers (Wikifier, GeoNames, Geotargets, GFK Products, and Google ProductsServices categories – also referred as Google Categories) and a central service that is responsible for unifying access to the underlying services.

A diagram illustrating the services involved is presented in Figure 6. In particular, the following components are identifiable:

1. ASIA, which is implemented as a component of Grafterizer, interacts with the central interlinking service sending the data to be reconciled and enriched together with a possible context.

2. The central inter-linking service (also referred to as ASIA backend) is responsible for calling and orchestrating the appropriate reconciliation and extension services among those
available. The extension can also be made using EW-Shopp core services, i.e., weather and events, which use date and position references as reference identifiers.

3. Reconciliation and extension services. In the figure, only three services are presented for space reasons.

The ASIA backend (whose internals are depicted in Figure 7) features four functionalities:

- **Full scale reconciliation.** This functionality supports the process of reconciliation and disambiguation from the services that actually perform it. Given a label and a set of reconciliation services of a certain type (e.g., spatial), this feature uses all these services to achieve a more accurate disambiguation. At the same time, it stores IDs from the different identifier spaces to enable the extension using the resources made available by all services.

- **Mapping via unique label.** The inter-linking service can itself be a reconciliation service using internal mapping structures. In particular, this feature translates unique labels (e.g., a code bar) from one system to another. For example, a label belonging to Google Categories can be converted into one of the DMOZ categories.

- **Reconciliation with lookup.** The central inter-linking service is able to query the individual reconciliation service and make it available to the user in a transparent manner.

- **Mapping via ID.** This is one of the central features of the interlinking service. In a nutshell, the service allows translating an ID of one system into the ID of another. To do this, the service must be able to load, maintain, update and manage a structure that maps identifiers that have meaning in different spaces. Using such a structure it is possible to use different services seamlessly.
As mentioned above, the presence of mapping structures that act as internal translators between identifiers belonging to different knowledge bases enables the inter-linking service to orchestrate and manage different reconciliation services. The component called Link Discover is the one in charge of identifying and managing these mappings. The three major functionalities of this component are:

1. Providing endpoint and storage space to users (or service providers) to load mappings. This mapping can be performed manually or by state-of-the-art software such as SILK\(^9\) and LIMES\(^10\).
2. Implementing an efficient translation service.
3. Creating and updating the mapping structure by adding mappings that can be identified through user requests.

At the moment the link discovery process relies on user-generated static mappings; nonetheless, we plan to implement state-of-the-art tools and techniques for the generation of such links in an automatized way.

As an example, the reconciliation of links between data sets for the Province of Milan (Italy) was created. We retrieved 135 entities, which could be cities, towns, and villages under the Province of Milan in GeoNames data set. For each of them, we extracted name, alternate names, administrative codes (adm1, adm2, adm3), population, latitude, longitude, elevation, country code, and time zone. For each of the GeoNames ID, we queried the Wikidata dataset to get NUTS identifier, Wikidata, Facebook Place, and Yahoo! WOEID IDs. We could find respectively 117, 2, 14, 4 identifiers for every

---

\(^9\) [http://silkframework.org/](http://silkframework.org/)

\(^10\) [http://aksw.org/Projects/LIMES.html](http://aksw.org/Projects/LIMES.html)
entity. With Wikidata IDs, we could query DBpedia to get the DBpedia identifier and we obtained 117 IDs (for each resource in Wikidata there exists the relative entity in DBpedia). Finally, to find the Google Code ID we use SILK to map the entities of Province of Milan to the Google’s Codes. We could generate 77 Google Codes as an exact match for the cities under the province of Milan.

So far, we have developed the service for fetching the information from GeoNames given a toponymy, and the coordinates. As every Knowledge Base points to GeoNames, the navigation through all of them becomes easy by following the mapping links. Although as a first step we considered only the links which are already mapped between data sets, the main challenge is to create and find all the mappings among the reference KBs as in the case of NUTS and Yahoo WOEID for the resource of Cerro Maggiore (see Table 3).

<table>
<thead>
<tr>
<th>Name</th>
<th>GeonamesID</th>
<th>Wikidata</th>
<th>DBpedia</th>
<th>Yahoo WOEID</th>
<th>Facebook Places</th>
<th>Google Code</th>
<th>NUTS</th>
<th>Latitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>Metropolitan city of Milan</td>
<td>gn:3173434</td>
<td>wdt:Q15121</td>
<td>dbr:Metropolitan City of Milan</td>
<td>NULL</td>
<td>NULL</td>
<td>08463</td>
<td>ITCAC</td>
<td>45.45186</td>
</tr>
<tr>
<td>SestoSan Giovanni</td>
<td>gn:6536522</td>
<td>wdt:Q43005</td>
<td>dbr:Sesto San Giovanni</td>
<td>12681991</td>
<td>NULL</td>
<td>1008491</td>
<td>NULL</td>
<td>45.53937</td>
</tr>
<tr>
<td>Vimodrone</td>
<td>gn:6541664</td>
<td>wdt:Q42419</td>
<td>dbr:Vimodrone</td>
<td>726242</td>
<td>NULL</td>
<td>1008507</td>
<td>NULL</td>
<td>45.51537</td>
</tr>
<tr>
<td>CerroMaggior e</td>
<td>gn:6539674</td>
<td>wdt:Q42507</td>
<td>dbr:Cerro Maggiore</td>
<td>NULL</td>
<td>109303599 096282</td>
<td>1008431</td>
<td>NULL</td>
<td>45.59527</td>
</tr>
</tbody>
</table>

**3.3.2 Workflow**

The purpose of this section is to describe the workflow for the use of inter-linking services, both in the phase of definition of the transformation and enrichment pipeline and in its execution by the big data runtime component.

Figure 8 is a graphic representation of the workflow regarding the design of a transformation and extension pipeline. As the dataset to be processed is often huge, it is loaded and managed in the corporate area. A corporate service generates a compact-sized sample of the data set and passes it on to EW-Shopp platform services. The transformation and enrichment service consists of two main components: Grafterizer, which supports the design and execution transformation operations on the dataset, and ASIA, a tool developed as a Grafterizer component, which allows the user to enrich the working data set by interacting with the reconciliation and extension services. In particular, ASIA frontend accesses the working data set through the Grafterizer API, allows the user to interactively perform entity linking on a column of his/her choice and provides a facility to explore and execute possible extensions. Finally, the operations performed by the user are collected in a transformation model that defines the pipeline to be performed on the original dataset in the corporate space with corporate facilities.
Figure 8. The data workflow to generate a transformation and enrichment pipeline.

Figure 9 shows how the transformation model (the reconciliation and extension pipeline) is performed on the original datasets. In contrast to the process of pipeline definition presented in Figure 8, the enrichment occurs without involving the user. The model includes algorithms and models trained to replicate the user’s choices based on his/her interaction with ASIA. In this way the Big Data Runtime component can engage with the reconciliation and extension services by making the disambiguation choices similar to those that the user would have made. Finally, the enriched data set is saved in an appropriate database to support the analytics components of the EW-Shopp platform.
It is worth noticing that both processes (the design of the transformation model at design time, and the execution of the transformation model on the corporate dataset at run time) make use of the reconciliation and extension services; therefore, they are part of the core data services.

### 3.4 Reconciliation and Extension Services

The aim of this section is to introduce the Reconciliation and Extension Services that are available at the time of writing. All the services presented below have been released under the GPL v3.0 licence and are available on GitHub12.

#### 3.4.1 Wikifier

In order to provide a multi-lingual general-purpose reconciliation service, we created a reconciliation service based on Wikifier13.

Given a textual document, Wikifier searches for mentions (name of Person, Organization, Place, etc.) in text and returns a list of potential entities matching each mention. Wikifier handles both DBpedia and WikiData entities; although, at the moment we only use DBpedia entities (i.e., we exploit the DBpedia identifiers space).

In more details, Wikifier Reconciliation Service (WRS) gets textual input (representing the content of a column of the working data set) pre-processed to ease and enhance the matching process. In particular, before a linking request is submitted to WRS, the text contained within each cell undergoes a splitting process that identifies the camel case pattern (e.g., “RealEstate” -> “Real Estate”). Then, the Wikifier Service is invoked through its public API (at moment using the default parameters configuration).

For each query, the WRS responds with a list featuring DBpedia entities IDs that with a certain probability refer to the entity mentioned in the query text. We use the PageRank parameter provided by Wikifier to assign a score to each matching entity.

At this time, it is not clear whether the PageRank is a meaningful enough index to be used to suggest a single entity per input cell to the user; for this reason, automatic matching for this service is disabled. Moreover, we are working to further improve WRS results because even if it is general-purpose and can harness multi-linguality, we noted that the performance of this service is not always good enough; this is mainly due to the shortness of text contained in cells to match. In fact, longer texts provide more “context”, which lead to better results. For the same reason, the Wikifier Service is not always able to auto-detect the correct language (e.g., given the “North Rhine-Westphalia” string as input, the auto-detected language is “Waray”, instead of “English”). At last, the

---

11 [http://www.gnu.org/licenses/](http://www.gnu.org/licenses/)
12 [https://github.com/UNIMIBInside/conciliator](https://github.com/UNIMIBInside/conciliator)
WikiWiki service does not return a “confidence” score for annotations; thus, it is not straightforward to assign a matching score for each cell (we use the PageRank value, but it depends on the neighbourhood graph built with the entities mentioned in the cell content). For all these reasons, we will work in the next months on improving the quality of the text sent to the service as well as testing possible context information to be send along.

3.4.2 GeoNames

The GeoNames Reconciliation Service (GRS) reconciles spatial mentions (i.e., toponyms) versus GeoNames entities (properly named “Features”).

This first version of this service relies on the GeoNames Search WebService (GSW)\(^\text{14}\), which returns a list of GeoNames features, given a toponym as input. The service is queried using the default parameters configuration (i.e., only the required fields are provided).

In particular, the GRS expects a toponym (e.g., “Milan”) or a list of toponyms (e.g., “Milan, Italy”) as input and returns ids for the GeoNames space of identifiers. Since addresses are not represented in GeoNames, this service cannot work with full addresses (e.g., “Viale Sarca 336, Milan, Lombardy, Italy”) but only with plain toponyms. Nonetheless, the GRS has the main benefits of being able to reconcile entities described in several languages.

Unlike the Wikifier-based reconciliation service, this service leverages a similarity measure called Jaro-Winkler distance\(^\text{15}\), which is fully supported by GeoNames. This makes it possible to implement the automatic matching feature by simply returning the best-scoring entity to the caller.

As a final note, since for developing purposes we are relying on the public GeoNames for the GRS, we are currently undergoing to GeoNames Terms and Conditions, which permit only 30,000 daily requests (limited to ~2000 per hour). Next versions of this service will use a local instance of GeoNames (i.e., a local RDF dump of GeoNames), implementing custom matching strategies.

3.4.3 Google Geotargets

This service reconciles geospatial entities versus Google Geotargets IDs. Google Geotargets dataset\(^\text{16}\) described 99,819 entities of 25 different types (see Table 4).

<table>
<thead>
<tr>
<th>Entity Type</th>
<th># entities</th>
<th>Entity Type</th>
<th># entities</th>
<th>Entity Type</th>
<th># entities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Airport</td>
<td>395</td>
<td>Department</td>
<td>222</td>
<td>Prefecture</td>
<td>47</td>
</tr>
<tr>
<td>Autonomous Community</td>
<td>22</td>
<td>District</td>
<td>588</td>
<td>Province</td>
<td>1035</td>
</tr>
</tbody>
</table>

\(^{14}\) [http://www.geonames.org/export/geonames-search.html]

\(^{15}\) [https://en.wikipedia.org/wiki/Jaro%E2%80%93Winkler_distance]

\(^{16}\) [https://developers.google.com/adwords/api/docs/appendix/geotargeting]
We do not consider Postal Codes and TV regions; in addition, because of the BC4 pilot scope, we only tested our service against spatial entities in Germany and Spain. Based on these filters, the final dataset used by this service contains 2,908 entities.

Given a toponym, this service returns a list of candidate Google Geotargets entities. The first release of this service does not handle neither list of toponyms, nor addresses. Since the considered datasets is small and publicly available, we use a local SPARQL endpoint where we ingested the RDF version of the considered subset of Google Geotargets. As response, this service returns the (internal) URIs of the Geotargets entities that probably describe the entity mentioned in a cell.

We use the Jaro-Winkler similarity between the cell content and the retrieved Geotarget entity name to assign a matching score. Thus, we can rank the candidates list and suggest to automatically match a cell when:

- more than one results is found, but the first result’s score is equal to 1.0 and
- there is only one result, which score is greater than a set threshold (we use 0.9 as default threshold).

At last, due to the nature of the underlying data set, the current version of this service handles only English toponyms.

### 3.4.4 Google ProductsServices

This service has been designed to reconcile category mentions against Google ProductsServices categories. This service is based on the Google ProductsServices dataset, which describes 3,180 categories, organized in taxonomy. Categories are provided only in English.

---

17 Next version will also include list of toponyms, exploiting the administrative hierarchy provided by Google Geotargets dataset

18 [https://developers.google.com/adwords/api/docs/appendix/productsservices.csv](https://developers.google.com/adwords/api/docs/appendix/productsservices.csv)
This service was implemented in a very similar way to what was done with the Google Geotargets service. Thus, this service is designed to accept a category as input\(^\text{19}\). The service is backed by a local SPARQL endpoint to access the ProductsServices dataset.

Since some datasets describe categories using the CamelCase pattern (e.g., JOT’s datasets), the service implements the search for categories whose entity name is reported in CamelCase. More in details, for each candidate category we compute the similarity considering both category’s official name and its CamelCase name, then we combine these similarities in a linear way.

Automatic matching strategies are the same as described for the Google Geotargets Reconciliation Service.

### 3.4.5 GFK Products

The Product Reconciliation Service (PRS) aims at reconciling product mentions (i.e., product name) against GfK’s Product Catalogue. GfK provides product data in tabular format. Based on the mapping described in [D1.2], the CSV data have been then transformed into RDF, making product data accessible through a local SPARQL endpoint.

The datasets ingested into the SPARQL endpoint are not fully multilingual, i.e., many attributes are not available in languages other than English. Based on this observation, the PRS support only English because the main attribute used within the reconciliation process (that is the product name) is only available in English. More in details, this service gets a label in input, which should relate a product name and reconciles the mentioned product against GfK’s Product Catalogue by finding similarities with GfK’s products name. Since GfK uses the “Type” attribute for describing both the article reference or the article name, we adopted a similarity measure (Jaro-Winkler) to find the best matching product.

No pre-processing is applied to the input text because normalizing the products name is a very complex task (e.g., Galaxy S IV is the same as Galaxy S 4, but both representations are used within the dataset).

Regarding the automatic matching strategy, we implemented the same solution used for the Google GeoTargets and Google ProductsServices Reconciliation services.

It should be noted that, since the “type” attribute provided by GfK contains some extra information other than the “canonical” product name (for example, the Galaxy S3 smartphone is represented as "I9300 Galaxy S III 16GB NFC LTE"), the matching performance achieved by the selected similarity measure cannot be not very high. In addition, even when the input label perfectly matches a product name, it could happen that the same name is present twice in the GfK Product Catalogue (for example, because the same product is sold in different countries, with different EAN codes) and no disambiguation strategies can be applied at this stage. Next versions of this service will implement

\(^{19}\) Next version will accept also a list of categories, where the first category represents the root category, while the last one represents the target category
3.4.6 OpenRefine Reconciliation Service API

As part of the work done in Work Package 3 we have identified the functionalities that services need to expose in order to be able to define a comprehensive and flexible API for reconciliation and extension. We have also analysed solutions on the market to get a full picture of the problem. We then discovered that there exists an API defined by OpenRefine to create reconciliation and extension services (we have borrowed the terms “Reconciliation” and “Extension” from the terminology proposed by OpenRefine). Using this API, it is possible to create services that can be immediately used in OpenRefine.

We believe that adopting OpenRefine Reconciliation API will bring three main benefits to the project:

1. The API is mature, well-defined, flexible, extensible and used every day by many services.
2. Following the specifications of OpenRefine allows the creation of services that may interest an audience of users, developers, and testers resulting in long-term sustainability of the project.
3. In addition, this choice allows the EW-Shopp ecosystem to access and use all services currently available for OpenRefine. In this way EW-Shopp can immediately exploit a set of mature reconciliation services that expose advanced functions on different knowledge bases.

Therefore, all services presented in this section have been developed following the OpenRefine Reconciliation Service API specification and are fully compatible with the latest version of OpenRefine. In fact, we tested our services against OpenRefine 3.0 BETA and OpenRefine 2.8.

In the remainder of this section the OpenRefine Reconciliation Service is briefly described. More details can be found on the GitHub page hosting the specifications.

In the OpenRefine API reconciliation service is RESTful and JSON-formatted. It consists of four endpoints, each of which fulfils a specific function, the most important of which are reconciliation and extension. Only the first one (reconciliation) is mandatory. The following table reports the endpoints along with a short description.

<table>
<thead>
<tr>
<th>Endpoint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Root URL (mandatory)</td>
<td>This base URL endpoint identifies the service and it is also used to request reconciliation operations.</td>
</tr>
<tr>
<td>&lt;root&gt;?suggest (optional)</td>
<td>This optional endpoint provides autocompletion functionality in</td>
</tr>
</tbody>
</table>

https://github.com/OpenRefine/OpenRefine/wiki/Reconciliation-Service-API
various points of the GUI.

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;root&gt;?preview (optional)</code></td>
<td>This optional endpoint provides a preview of the reconciled items.</td>
</tr>
<tr>
<td><code>&lt;root&gt;?extend (optional)</code></td>
<td>This optional endpoint provides the application with the ability to add columns from reconciled values based on the properties of the entity in the underpinning knowledge base.</td>
</tr>
</tbody>
</table>

### 3.4.6.1 Root endpoint

The root endpoint supports two usage scenarios:

1. If called without parameters, the service returns an object called **service metadata**. A table describing the field of the service metadata object as well as an example of such an object are reported below. The pictures reported in this section are taken from the OpenRefine Reconciliation Service API.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;name&quot;</td>
<td>This is the name of the service, which will be used to display the service in the reconciliation menu;</td>
</tr>
<tr>
<td>&quot;identifierSpace&quot;</td>
<td>This is an URI for the type of identifiers returned by the service;</td>
</tr>
<tr>
<td>&quot;schemaSpace&quot;</td>
<td>This is an URI for the type of types understood by the service.</td>
</tr>
<tr>
<td>&quot;view&quot;</td>
<td>View is an object associated with a template URL to view a given entity from its identifier: &quot;view&quot;: {&quot;url&quot;:&quot;<a href="http://example.com/object/%7B%7BID%7D%7D%22%7D">http://example.com/object/{{ID}}&quot;}</a></td>
</tr>
</tbody>
</table>

Other fields (optional) They are used to specify the URLs for the other endpoints (suggest, preview and extend).

```json
{
  "name": "Wikidata Reconciliation for OpenRefine (en)",
  "identifierSpace": "http://www.wikidata.org/entity/",
  "schemaSpace": "http://www.wikidata.org/prop/direct/",
  "view": {
    "url": "https://www.wikidata.org/wiki/{{id}}"
  },
  "defaultTypes": [],
  "preview": {
    ...
  },
  "suggest": {
    ...
  },
  "extend": {
    ...
  }
}
```
2. If it is called with the **queries** parameter (followed by a list of json object containing a single query definition, it executes a set of batch queries and returns the results to the caller. A table listing the fields of the query object along with their description and a full example of such an object are reported below.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;query&quot;</td>
<td>A string to search for. Required.</td>
</tr>
<tr>
<td>&quot;limit&quot;</td>
<td>An integer to specify how many results to return. Optional.</td>
</tr>
<tr>
<td>&quot;type&quot;</td>
<td>A single string, or an array of strings, specifying the types of result e.g., person, product, ... The actual format of each type depends on the service (e.g., &quot;Q515&quot; as a Wikidata type). Optional.</td>
</tr>
<tr>
<td>&quot;type_strict&quot;</td>
<td>A string, one of &quot;any&quot;, &quot;all&quot;, &quot;should&quot;. Optional.</td>
</tr>
<tr>
<td>&quot;properties&quot;</td>
<td>Array of json object literals. Optional</td>
</tr>
</tbody>
</table>

```json
{
    "query" : "Ford Taurus",
    "limit" : 3,
    "type" : "Q3231690",
    "type_strict" : "any",
    "properties" : [
        { "p" : "P571", "v" : 2009 },
        { "pid" : "P176" , "v" : { "id" : "Q20827633" } }
    ]
}
```

### 3.4.6.2 Extend endpoint

The API that allows the extension is accessible through a special endpoint that is exposed using the metadata service object. In particular, this can contain two fields, both optional:

- "propose_properties" provides an endpoint used to suggest properties to fetch. If this field is not provided, no property will be suggested to the caller.
- "property_settings" stores the specification of a form where the user will be able to configure how a given property should be fetched.

Example of service metadata:

```json
"extend": {
    "propose_properties": {
        "service_url": "https://tools.wmflabs.org/openrefine-wikidata",
        "service_path": "/en/propose_properties"
    },
```
The consumer calls the root URL endpoint with a JSON object in the `extend` parameter, containing the following fields:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;IDs&quot;</td>
<td>This is a list of strings, each of which being an identifier of a record as returned by the reconciliation method. These are the records whose properties should be retrieved.</td>
</tr>
<tr>
<td>&quot;properties&quot;</td>
<td>This is a list of JSON objects. They specify the properties to be fetched for each item.</td>
</tr>
</tbody>
</table>

Example:

```
{
  "ids": [
    "Q7205598",
    "Q218765",
    "Q845632",
    "Q5661356"
  ],
  "properties": [
    {
      "id": "P856"
    },
    {
      "id": "P159"
    }
  ]
}
```

The service returns a JSON response formatted as follows:

- "meta" contains a list of column metadata. The order of the properties must be the same as in the query.
- "rows" contains an object. Its keys must be exactly the record IDs (IDs) passed in the query. The value for each record ID is an object representing a row for that ID.

More details can be found in the Data Extension API web page\(^{21}\) which is also the source for the images used in this section.

3.5 Reconciliation Services development

All services presented in this chapter are developed using conciliator\textsuperscript{22}, a Java/Spring framework specifically created to streamline the creation of reconciliation services implementing the OpenRefine Reconciliation and Extension API (source code released under GPL 3.0 license). In addition, conciliator already provides multi-thread capabilities (on demand), and implements caching solutions.

According to Conciliator Licence the source code of the current version of the reconciliation services will be released under GPL 3.0 license, however:

- The underlying datasets (e.g., GfK products datasets and geospatial mappings) and API keys (e.g., Wikifier API key) won’t be released as open.
- Ad-hoc matching algorithms will be packed in libraries subject to suitable licence schemes.

Chapter 4 Analytics Services

This chapter presents the current state of the analytics services as part of the EW-Shopp platform. Section 4.1 introduces the background needed for understanding the rest of the chapter, including a short description of the QMiner analytics framework in section 4.1.1. The core of the chapter follows the typical analytics workflow by explaining how to load the data in section 4.2, how to build a model in section 4.3 and how to deploy the model so it can receive prediction requests and return predictions in section 4.4. The chapter concludes with a short note on future work for the development of the analytics services.

4.1 Introduction and Terminology

The analytics stage stands after the transformation and linking stage in the EW-Shopp platform. Analytical services consume the data prepared by the transformation and linking services and use it to analyse the business processes described by the data.

The result of the analysis can in principle be insight into the nature and dynamics of these processes or an operational model which can be used to power new services. For example, we can aggregate the pricing information for different stores participating on a price comparison platform and plot them to observe their pricing strategies (insight) or use data mining methodology to build a predictive model which uses the weather forecast to predict when summer temperatures will drive up the demand for air conditioning units (operational). The first result can be inspected by management of a store to help plan their quarterly sales, whereas the second can be integrated directly into a store website to offer additional discounts when their effect is maximal.

\textsuperscript{22} https://github.com/codeforkjeff/conciliator
There exist different methodologies to perform such analyses and an abundance of software tools, both free and proprietary. The core of the EW-Shopp analytics is built using the QMiner\textsuperscript{23} data analytics platform developed and maintained at JSI. Similar results could also be obtained using some other data mining library or loading it into an analytics platform such as Weka\textsuperscript{24} or R\textsuperscript{25}. Businesses might also have commercial business intelligence software they are used to using. One of the advantages of the loosely-coupled architecture of the EW-Shopp platform is that substituting the analytics stage with some other tool of choice is trivial. Pilots in business cases BC3 and BC4 described in [2], which use statistical modelling methods in the R platform, demonstrate this.

As input the analytics stage receives the transformed and linked dataset prepared for analysis. This means, for example, that sales from a particular store are linked to appropriate weather data based on store location and sale time. This data can be stored in a database or (typically during development) dumped in a file. The output of the analytics is either some representation of the extracted information (e.g. a table with processed values or a chart) or a model. The latter is typically a mathematical formula or structure encoded in some language that can be read by a tool to run it on some new data.

All the operations in the API are entirely configured using JSON-formatted settings objects. This, for example, includes input data locations and database access credentials as well as specification of the modelling algorithm and its parameters. Such a unified representation of the entire process gives us the flexibility to, for example: develop a model on a sample dataset offsite and then transfer the specification to the QMiner instance on the corporate server, to build the full model; automatically spawn instances of analytical services in virtual machines or Docker containers; store several different model specifications in a database along with their performance metrics etc.

4.1.1 QMiner Overview

To fully understand the API descriptions in the rest of the chapter, some understanding of QMiner functionality is helpful. QMiner is an analytics platform with strong support for different data types, such as structured (e.g. shopping baskets, visit logs), textual (e.g. news articles, emails), time series (e.g. sensors, financial indicators), networks (e.g. social graphs).

QMiner architecture is built around a NoSQL-like storage layer optimized for fast processing and feature extraction. Note that it was not designed for long-term storage, such as a more generic database, which is why it is unsuitable as the central platform storage solution. It has a comprehensive core library of supervised and unsupervised machine learning algorithms implemented in C++ for efficiency and scalability. For ease of use, these core algorithms are wrapped in a JavaScript library which enables fast prototyping and simple integration into other processes. The architecture is shown in Figure 10.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
\textbf{EW-Shopp} & \textbf{GA number: 732590} & \textbf{H2020-ICT-2016-2017/H2020-ICT-2016-1} \\
\hline
\end{tabular}
\end{table}

\textsuperscript{23} http://qminer.github.io
\textsuperscript{24} http://www.cs.waikato.ac.nz/ml/weka
\textsuperscript{25} http://www.r-project.org
Performing an analytics task follows a typical workflow presented in Figure 11. Data is first loaded into the indexed storage, from there it is transformed into a mathematical representation through feature extraction and finally this representation is used to run some analytical algorithm which produces a model. The middle transformation step is needed as the algorithms need the data represented as vectors of numbers. For an example of such a transformation, let’s assume we have a sales dataset with the date of each sale recorded (among other information). If we want to analyse the sales volume then one important factor we should consider is the day of the week (Monday, Tuesday …). To represent this a typical approach would be to add seven columns to the data matrix, one for each day, and set the column corresponding to the correct day to 1 and the rest to 0. This form is highly inefficient for storage, which is why this transformation is not performed by the EW-Shopp transformation services.

There is another reason why it is useful for some transformations to be performed at this stage, even if they might be possible in the transformation stage. Performing analytics is by nature an explorative process. When developing a model, we typically do not know in advance what kind of featurization will be optimal. For example, when predicting sales of air conditioning units – how many days’ weather is relevant? Do people start buying them after two hot days, three hot days or does the whole week need to be hot? Aggregating the weather data over all these different time ranges in advance and storing that in the database makes little sense. It is much more efficient if we can compute the different aggregations on the fly when experimenting with models. Once an optimal transformation is discovered, we can move this operation to the transformation stage if necessary or desired.
Figure 11. Data analysis workflow in QMiner.

The following two sections describe APIs used for the loading step (section 4.2) and transformation and analysis (both in section 4.3).

### 4.2 Data Loading

The data loading module transports data from different sources into the QMiner database. Three types of sources are currently supported: ArangoDB\(^{26}\), MariaDB\(^{27}\) and a comma-separated values (CSV) file. The entire transfer operation is specified in a configuration JSON which is given to the module as input.

#### 4.2.1 Configuration JSON specification

The configuration file is provided to move data from source to a destination QMiner database. First, the process connects to the source and destination. Second, the queries are executed in the given order in the "queries" list. Each query collects records from one source and transfers them into a destination database store. Finally, the connections are closed.

The configuration is structured into four top-level fields:

```json
{
    "source": { ... },
    "destination": { ... },
    "queries": [
        { ... },
        { ... },
        ...
    ],
    "misc": { ... }
}
```

These top-level fields are described in the following table. Individual field contents are explained in the rest of the section.

---

\(^{26}\) [http://www.arangodb.com](http://www.arangodb.com)

\(^{27}\) [http://mariadb.org](http://mariadb.org)
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>source</td>
<td>Object</td>
<td>Yes</td>
<td>Parameters of the data source.</td>
</tr>
<tr>
<td>destination</td>
<td>Object</td>
<td>Yes</td>
<td>Parameters of the data destination.</td>
</tr>
<tr>
<td>queries</td>
<td>List</td>
<td>Yes</td>
<td>List of queries that will be executed.</td>
</tr>
<tr>
<td>misc</td>
<td>Object</td>
<td>No</td>
<td>Miscellaneous options.</td>
</tr>
</tbody>
</table>

**Source**

The source parameters vary depending on the source type.

**ArangoDB/MariaDB**

```json
"source": {
  "type": str,
  "host": str,
  "database": str,
  "user": str,
  "password": str
}
```

**Csv file**

```json
"source": {
  "type": str,
  "dir": str,
  "filename": str,
  "custom_fn_path": str
}
```

**Parameter description:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Required</th>
<th>Source type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>String</td>
<td>Yes</td>
<td>All</td>
<td>Type of source. Possible values: [&quot;Csv&quot;, &quot;ArangoDB&quot;, &quot;MariaDB&quot;].</td>
</tr>
<tr>
<td>host</td>
<td>String</td>
<td>Yes</td>
<td>MariaDB, ArangoDb</td>
<td>The hostname of the source database.</td>
</tr>
<tr>
<td>database</td>
<td>String</td>
<td>Yes</td>
<td>MariaDB, ArangoDb</td>
<td>Name of the source database.</td>
</tr>
<tr>
<td>user</td>
<td>String</td>
<td>Yes</td>
<td>MariaDB, ArangoDb</td>
<td>The user to authenticate as.</td>
</tr>
<tr>
<td>password</td>
<td>String</td>
<td>Yes</td>
<td>MariaDB, ArangoDb</td>
<td>The password of that user.</td>
</tr>
<tr>
<td>dir</td>
<td>String</td>
<td>No</td>
<td>Csv</td>
<td>Directory in which CSV file is placed.</td>
</tr>
</tbody>
</table>
filename | String | Yes | Csv | Filename or filepath to CSV file.
---|---|---|---|---
custom_fn_path | String | No | Csv | Filename or filepath to file with custom functions. (see explanation in section 4.2.1.2)

**Destination**

Specification of the destination QMiner database.

```
"destination": {  
  "type": str,  
  "db_path": str,  
  "mode": str  
}
```

Parameter description:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>String</td>
<td>Yes</td>
<td>Type of source/destination. Possible values: [&quot;Csv&quot;].</td>
</tr>
<tr>
<td>db_path</td>
<td>String</td>
<td>Yes</td>
<td>The path to QMiner db directory.</td>
</tr>
<tr>
<td>mode</td>
<td>String</td>
<td>No</td>
<td>Base access mode. Default is &quot;openReadOnly&quot;. Access modes are the same as described in QMiner base constructor (^{28}), with additional &quot;extend&quot; option, which appends new schemas, defined in queries, to existing QMiner db.</td>
</tr>
</tbody>
</table>

**Queries**

Parameter “queries” has a list of queries where a query is defined with the following parameters.

```
"queries": [  
  
    {  
      "name": str,  
      "query_src": {  
        "query": str,  
        "placeholder_mapping": str,  
        "read_line_fn": str,  
        "read_line_fn_args": [...]  
      },  
      "use_query": bool,  
      "use_schema": bool,  
      "mapping": {  
        "schema": obj  
      }  
    }  
]
```

[^28]: https://rawgit.com/qminer/qminer/master/nodedoc/module-qm.html#~BaseConstructorParam
Parameter description:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>Yes</td>
<td>Name of the query. Note: When using ArangoDB as a source and &quot;query_src&quot; is not provided, this name is used as collection name to get all documents.</td>
</tr>
<tr>
<td>use_query</td>
<td>Boolean</td>
<td>Yes</td>
<td>Flag to use this query.</td>
</tr>
<tr>
<td>use_schema</td>
<td>Boolean</td>
<td>Yes</td>
<td>When &quot;mode&quot; is set to &quot;createClean&quot;, &quot;create&quot; or &quot;extend&quot;, this flag indicates to use schema, defined in &quot;mapping.schema&quot;, to build a new store in QMiner base.</td>
</tr>
<tr>
<td>query_src</td>
<td>Object</td>
<td>Yes</td>
<td>Object with parameters to query source. (Optional for csv sources)</td>
</tr>
<tr>
<td>query_src.query</td>
<td>String</td>
<td>Yes</td>
<td>[ArangoDB/MariaDB only] Query string to query source or destination. ArangoDB query language or SQL string.</td>
</tr>
<tr>
<td>query_src.placeholder_mapping</td>
<td>String</td>
<td>No</td>
<td>[MariaDB only] Values from collected records to be mapped to values in a query.</td>
</tr>
<tr>
<td>query_src.read_line_fn</td>
<td>String</td>
<td>No</td>
<td>[csv only] Name of the custom function. (See explanation in section 4.2.1.1)</td>
</tr>
<tr>
<td>query_src.read_line_fn_args</td>
<td>List</td>
<td>No</td>
<td>[csv only] Additional arguments to custom function. (See explanation in section 4.2.1.1)</td>
</tr>
<tr>
<td>mapping</td>
<td>Object</td>
<td>Yes</td>
<td>Object with schema field. (see next row)</td>
</tr>
<tr>
<td>mapping.schema</td>
<td>List</td>
<td>Yes</td>
<td>Store schema definitions to be created in QMiner base if flag &quot;use_schema&quot; is set. Also used for mapping data from various sources to QMiner base. See section 4.2.1.1 for details.</td>
</tr>
</tbody>
</table>

**Misc**

Currently, miscellaneous options only control the logging verbosity.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
</table>

EW-Shopp        GA number: 732590        H2020-ICT-2016-2017/H2020-ICT-2016-1
4.2.1 Mapping

The structure of the "schema" field is the same as in QMiner store schema definition extended with mapping information. Each field object is wrapped in the name of the source column. For example, let's say we would like to map string values from "EventID" column in the source to a QMiner record field "eventID". The standard QMiner field specification object is wrapped in an object with the source column name as follows:

```
{
  "EventID": {
    "name": "eventID",
    "type": "string"
  }
}
```

4.2.1.2 Csv special considerations

The CSV format is popular for data exchange due to its simplicity. As such it is commonly used when dumping dataset samples. In the project it was mostly used in the initial stages for development and is not used in deployment scenarios. Because of its simple structure, some special logic is typically needed to properly parse its contents (e.g. correctly parsing date strings). The loading API supports extension with custom parsing functions which can be referenced in the specification JSON. This is, of course, a crude solution and not one that is recommended for use in production, but has proven invaluable for speeding up development.

By default, when querying CSV file, each line is read and destination "query" is executed to store the record. In order to filter or edit CSV line before storing it into destination base, you can define custom functions. Set "read_line_fn" to name of the custom function. Furthermore, "custom_fn_path" must be set in the "source" parameter, showing the path to the file with custom functions.

A custom function can be defined in two ways – with returning record or with a query to the destination database. In case of passing two parameters, it is assumed, that function returns processed record. The first parameter is a list of values of the current CSV line and the second parameter is a list of additional arguments passed via "read_line_fn_args". On the contrary, when a custom function has three parameters, the first parameter is a connection to the destination database. Therefore, a custom function must store CSV line to the destination database and the "query_dst" field is ignored.

4.2.2 Examples

https://rawgit.com/qminer/qminer/master/nodedoc/module-qm.html#~SchemaFieldDef
CSV to Qminer base

A simple example to load weather data from CSV file to Qminer base.

```json
{
    "source": {
        "type": "Csv",
        "dir": "../data/",
        "filename": "slovenia-2014-2017_qminer.tsv"
    },
    "destination": {
        "type": "QminerDB",
        "db_path": "/dbWeather/",
        "mode": "createClean"
    },
    "queries": [
        {
            "name": "Example Schema",
            "use_query": true,
            "use_schema": true,
            "mapping": {
                "schema": [
                    {
                        "name": "weather",
                        "fields": [
                            {
                                "param": {
                                    "name": "Param",
                                    "type": "string"
                                }
                            },
                            {
                                "timestamp": {
                                    "name": "Timestamp",
                                    "type": "datetime"
                                }
                            },
                            {
                                "dayOffset": {
                                    "name": "DayOffset",
                                    "type": "int"
                                }
                            },
                            {
                                "region": {
                                    "name": "Region",
                                    "type": "int"
                                }
                            },
                            {
                                "value": {
                                    "name": "Value",
                                    "type": "float"
                                }
                            }
                        ]
                    }
                ]
            }
        }
    ]
}
```
ArangoDB to Qminer base

A similar example using ArangoDB as source.

```json
{
  "source": {
    "type": "ArangoDB",
    "host": "http://127.0.0.1:8529",
    "database": "_system",
    "user": "root",
    "password": "root"
  },
  "destination": {
    "type": "QminerDB",
    "db_path": "./dbExample/",
    "mode": "open"
  },
  "queries": [
    {
      "name": "SloveniaWeather",
      "query_src": {
        "query": "FOR forecast IN `slovenia-weather` RETURN forecast"
      },
      "use_query": true,
      "use_schema": true,
      "mapping": {
        "schema": {
          "clear": false,
          "name": "SloveniaWeather",
          "fields": [
            {
              "Param": {
                "name": "param",
                "type": "string",
                "shortstring": true
              }
            },
            {
              "Value": {
                "name": "value",
                "type": "float"
              }
            },
            {
              "Timestamp": {
                "name": "timestamp",
                "type": "datetime"
              }
            }
          ]
        }
      }
    }
  ]
}
```
4.3 Model Building

The model building module transforms the dataset in a QMiner database into a set of features and builds a model predicting some target variable. It combines feature extraction and model building into a single modelling pipeline. The pipeline is specified in a configuration JSON which is given to the module as input.

4.3.1 Configuration JSON specification

The modelling pipeline consists of two main parts – feature transformation and model building. Feature transformation builds new features from those already in the dataset. As explained in the introduction in section 4.1, these are transformations for which it is more practical to be performed during modelling as it simplifies exploration. The main example for this is aggregation of weather data. After the transformation a model is built by specifying the target feature, the features to use and the model type.

The configuration file top-level structure is:

```json
{
    "Region": {
        "name": "region",
        "type": "int"
    },
    "DayOffset": {
        "name": "dayOffset",
        "type": "int"
    },
    "joins": [],
    "keys": [
        {
            "field": "timestamp",
            "type": "linear"
        },
        {
            "field": "param",
            "type": "value"
        }
    ]
}
```

"misc": {
    "verbose": true
}
}
Parameter description:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>transform</td>
<td>List</td>
<td>Yes</td>
<td>List containing all the feature transformers that should be executed in the build phase.</td>
</tr>
<tr>
<td>build_model</td>
<td>Object</td>
<td>Yes</td>
<td>Specification of the model building operation.</td>
</tr>
<tr>
<td>build_model.target</td>
<td>Object</td>
<td>Yes</td>
<td>Target feature extractor. Should return a single feature.</td>
</tr>
<tr>
<td>build_model.features</td>
<td>List</td>
<td>Yes</td>
<td>List of feature extractors for the model to use (all should store the result in the same database).</td>
</tr>
<tr>
<td>build_model.model</td>
<td>Object</td>
<td>Yes</td>
<td>Model to be trained.</td>
</tr>
</tbody>
</table>

4.3.1.1 Feature transformers

Feature transformers build new features from existing ones. The only transformer currently supported is the weather_transformer:

```json
{
   "transformer": "weather_transformer",
   "params": {
      "inputDb": str,
      "outputDb": str,
      "forecastOffsets": [int, int ...],
      "ranges": {
         "date": [int, int], [int, int], ...
      }
   }
}
```

Parameter description:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inputDb</td>
<td>String</td>
<td>Yes</td>
<td>Location of the QMiner database containing raw weather data.</td>
</tr>
<tr>
<td>outputDb</td>
<td>String</td>
<td>Yes</td>
<td>Location of the QMiner database where result is stored.</td>
</tr>
<tr>
<td>forecastOffsets</td>
<td>List</td>
<td>Yes</td>
<td>List of weather forecast offsets that should be used.</td>
</tr>
</tbody>
</table>
4.3.1.2 Feature extractors

There are two feature extractors currently supported: feature_selector and weather_feature_selector.

**feature_selector**

Select a subset of features.

```
{
  "extractor": "feature_selector",
  "params": {
    "inputDb": str,
    "outputDb": str,
    "features": [str, str ...],
  }
}
```

Parameter description:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inputDb</td>
<td>String</td>
<td>Yes</td>
<td>Location of the QMiner database containing the selected features.</td>
</tr>
<tr>
<td>outputDb</td>
<td>String</td>
<td>Yes</td>
<td>Location of the QMiner database where result is stored.</td>
</tr>
<tr>
<td>features</td>
<td>List</td>
<td>Yes</td>
<td>List of selected feature names.</td>
</tr>
</tbody>
</table>

**weather_feature_selector**

Select a subset of weather features built with weather_transformer. This extractor performs the same functionality as the feature_selector, but can specify ranges of features in the same way as the weather_transformer so that the weather features need not be listed one by one.

```
{
  "extractor": "weather_feature_selector",
  "params": {
    "inputDb": str,
    "outputDb": str,
    "forecastOffsets": [int, int ...],
    "ranges": {
      "date": [ [int,int], [int,int], ... ],
      "time": [ [int,int], [int,int], ... ],
    }
  }
}
```

Parameter description:
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inputDb</td>
<td>String</td>
<td>Yes</td>
<td>Location of the QMiner database containing selected weather data.</td>
</tr>
<tr>
<td>outputDb</td>
<td>String</td>
<td>Yes</td>
<td>Location of the QMiner database where result is stored.</td>
</tr>
<tr>
<td>forecastOffsets</td>
<td>List</td>
<td>Yes</td>
<td>List of weather forecast offsets that should be used.</td>
</tr>
<tr>
<td>ranges/date</td>
<td>List</td>
<td>Yes</td>
<td>List of date ranges.</td>
</tr>
<tr>
<td>ranges/time</td>
<td>List</td>
<td>Yes</td>
<td>List of time ranges.</td>
</tr>
</tbody>
</table>

### 4.3.1.3 Models
Two types of models are supported – an SVM classifier and an SVM regressor. Both have the same specification structure.

```json
{
  "model": str,
  "params": {
    "inputDb": str,
    "SVMParam": obj,
    "model_filename": str
  }
}
```

Parameter description:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>model</td>
<td>String</td>
<td>Yes</td>
<td>Model type. Possible values: [&quot;SVC&quot;, &quot;SVR&quot;].</td>
</tr>
<tr>
<td>inputDb</td>
<td>String</td>
<td>Yes</td>
<td>Location of the QMiner database containing selected features to use.</td>
</tr>
<tr>
<td>SVMParam</td>
<td>String</td>
<td>Yes</td>
<td>SVM model parameters as specified in the QMiner documentation.</td>
</tr>
<tr>
<td>model_filename</td>
<td>String</td>
<td>Yes</td>
<td>File to dump model.</td>
</tr>
</tbody>
</table>

### 4.3.2 Example
An example of building a regression model to predict air conditioning pageviews from weather data.

```json
{
  "transform": [
    {
      "transformer": "weather_transformer",
      "SVMParam": ...
    }
  ]
}
```

[30](https://rawgit.com/qminer/qminer/master/nodedoc/module-analytics.html#SVMParam)
"params": {
  "inputDb": "weatherDb",
  "outputDb": "weatherFeaturesDb",
  "forecastOffsets": [-6, -5, -4, -3, -2, -1, 0],
  "ranges": {
    "date": [
      [-4, 0],
      [-3, 0],
      [-2, 0],
      [-1, 0]
    ],
    "time": [
      [0, 23],
      [6, 12],
      [12, 18]
    ]
  }
},
"build_model": {
  "target": {
    "extractor": "feature_selector",
    "params": {
      "inputDb": "productsDb",
      "outputDb": "targetAndFeaturesDb",
      "features": [ "ac_pageviews" ]
    }
  },
  "features": [
    {
      "extractor": "weather_feature_selector",
      "params": {
        "inputDb": "weatherFeaturesDb",
        "outputDb": "targetAndFeaturesDb",
        "forecastOffset": -6,
        "ranges": {
          "date": [
            [-4, 0],
            [-3, 0],
            [-2, 0],
            [-1, 0]
          ],
          "time": [
            [0, 23],
            [6, 12],
            [12, 18]
          ]
        }
      }
    }
  ],
  "model": {
    "model": "SVR",
    "params": {
      "inputDb": "targetAndFeaturesDb",
      "SVMParam": {
        "algorithm": "LIBSVM",
      }
    }
  }
}
4.4 Prediction

Having built the model, we can use it for prediction on new data. A light-weight approach which offers a lot of flexibility for integration into other services is to set up a REST service which receives a new example and returns a prediction from the model. Both the server setup as well as the request and reply are specified by JSON.

4.4.1 Prediction Server Setup

The prediction server receives HTTP POST requests with JSON data describing examples and returns predictions the model produces for these examples. Since the requests may contain sensitive information we support encrypted connections over SSL. The configuration JSON is:

```
{
  "model_filename": str,
  "host": str,
  "port": int,
  "security": {
    "key_filename": str,
    "certificate_filename": str
  }
}
```

Parameter description:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>model_filename</td>
<td>String</td>
<td>Yes</td>
<td>Location of the file containing the serialized QMiner model to load.</td>
</tr>
<tr>
<td>host</td>
<td>String</td>
<td>Yes</td>
<td>Hostname for the server.</td>
</tr>
<tr>
<td>port</td>
<td>Integer</td>
<td>Yes</td>
<td>Port for the server.</td>
</tr>
<tr>
<td>security</td>
<td>Object</td>
<td>No</td>
<td>Credentials for SSL. If not present, the connections are unencrypted.</td>
</tr>
<tr>
<td>security.key_filename</td>
<td>String</td>
<td>No</td>
<td>File containing the server private key.</td>
</tr>
<tr>
<td>security.certificate_filename</td>
<td>String</td>
<td>No</td>
<td>File containing the server certificate.</td>
</tr>
</tbody>
</table>
4.4.2 Prediction Request and Reply

Prediction request contains a list of examples to run the models on. The examples in the data field should be structured according to the schema used to build the model (see section 4.3.1).

```
{
    "request": {
        "data": [ obj, obj, ... ]
    }
}
```

Parameter description:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>request</td>
<td>Object</td>
<td>Yes</td>
<td>A field with the prediction request data.</td>
</tr>
<tr>
<td>data</td>
<td>List</td>
<td>Yes</td>
<td>A list of examples to run the prediction model on. Should be structured according to model schema.</td>
</tr>
</tbody>
</table>

The reply returns a list of predictions the model produced.

```
{
    "reply": {
        "data": [...]
    }
}
```

Parameter description:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reply</td>
<td>Object</td>
<td>Yes</td>
<td>Field with the prediction data.</td>
</tr>
<tr>
<td>data</td>
<td>List</td>
<td>Yes</td>
<td>A list of predictions produced by the model in the same order as the input examples. The type of the predictions depends on the model. A regression model returns numerical values, whereas a classification model may return string values.</td>
</tr>
</tbody>
</table>

4.4.3 Example

A simplified example for a model predicting pageviews for air conditioning units form average daily temperature.

**Server setup:**

```
{
    "model_filename": "AC_pageviews_regressor.model",
    "host": "localhost",
}
```
"port": "1234",
"security": {
    "key_filename": "security/privkey.pem",
    "certificate_filename": "security/fullchain.pem"
}
}

Request:
{
    "request": [  
        "data": [  
            { "Date": "2015-07-01", "Mean_T_daterange_0_0_timerange_0_23": 28.3},  
            { "Date": "2015-07-02", "Mean_T_daterange_0_0_timerange_0_23": 31.7}  
        ]
    ]
}

Reply:
{
    "reply": [  
        "data": [  
            { "ac_pageviews": 76 },  
            { "ac_pageviews": 103}  
        ]
    ]
}

4.5 Future Work

There are many possibilities for extension of the current EW-Shopp analytics framework. The most immediate one being the addition of more feature transformers and selectors. For example, transformers and selectors more directly supporting event data. The current state was developed to support the pilot services described in deliverable D4.2 [2]. As those services develop and expand (both in functionality and considered data types), so will the analytics services.

Chapter 5  Conclusion

This document presents the transformation, linking and analytics services in the EW-Shopp platform. Each set of services is covered in a separate chapter, ordered as they are in the typical workflow. Data is first ingested and transformed into the format suitable for processing, it is then interlinked, either internally or to external reference datasets, to connect data points between each other or to relevant context data; finally, the prepared data is passed through an analytics algorithm, building a model that can be used to produce predictions on new data.
In principle, all the services in this deliverable do not have any specific hardware requirements and can be installed and run on a run-of-the-mill laptop. Of course, this is highly dependent on the data volume to process and the desired service latency. Most requirements (for both processing power as well as memory) increase linearly with data size, but there are exceptions. For example, using a certain set of parameters for the model learning algorithm can make the model learning complexity quadratic in the number of data points. During their runtime the pilot services described in deliverable D4.2 [2] should give more insight into the hardware needed by these services in an operational setting.

With the exception of the SSL security credentials in the prediction service server specification, there is no explicit consideration of security measures. Security is by assumption decoupled from the application protocols the services in this deliverable use. For example, database access is secured by using an encrypted VPN connection (as described in [1]). If need be, the services could also be secured using the Simple Authentication and Security Layer\(^{31}\).

These services are still in active development. As we are at the mid-point of the project, this document represents a current snapshot of the platform. The chief goal of the development efforts so far was to support the development of the pilot services (see D4.2 [2]). As lessons are learned during pilots’ lifetime and we move towards the production deployment in the second half of the project, these services will continue to evolve. All significant upgrades of the services will be included in the report describing the final state of the platform.

References


\(^{31}\) https://tools.ietf.org/html/rfc4422