



Project Number: **770299**

NewsEye:
A Digital Investigator for Historical Newspapers

Research and Innovation Action
Call H2020-SC-CULT-COOP-2016-2017

D5.6: Personal Research Assistant: Investigator (c) (final)

Due date of deliverable: M45 (31 January 2022)

Actual submission date: 31 January 2022

Start date of project: 1 May 2018

Duration: 45 months

Partner organization name in charge of deliverable: UH-CS

Project co-funded by the European Commission within Horizon 2020		
Dissemination Level		
PU	Public	PU
PP	Restricted to other programme participants (including the Commission Services)	-
RE	Restricted to a group specified by the Consortium (including the Commission Services)	-
CO	Confidential, only for members of the Consortium (including the Commission Services)	-

Revision History

Document administrative information	
Project acronym:	NewsEye
Project number:	770299
Deliverable number:	D5.6
Deliverable full title:	Personal Research Assistant: Investigator (c) (final)
Deliverable short title:	Personal Research Assistant: Investigator (final)
Document identifier:	NewsEye-T51-D56-Investigator-c-final-Submitted-v6.0
Lead partner short name:	UH-CS
Report version:	V6.0
Report preparation date:	31.01.2022
Dissemination level:	PU
Nature:	Report
Lead author:	Lidia Pivovarova (UH-CS)
Co-authors:	Simo Linkola, Leo Leppänen and Hannu Toivonen (UH-CS)
Internal reviewers:	Eva Pfanzelter (UIBK-ICH), Axel Jean-Caurant (ULR)
Status:	<input type="checkbox"/> Draft
	<input type="checkbox"/> Final
	<input checked="" type="checkbox"/> Submitted

The NewsEye Consortium partner responsible for this deliverable has addressed all comments received, making changes as necessary. Changes to this document are detailed in the change log table below.

Change Log

Date	Version	Editor	Summary of changes made
30/03/2021	0.1	Lidia Pivovarova (UH-CS)	First full draft
31/03/2020	1.0	Hannu Toivonen (UH-CS)	WP5 leader's check
24/04/2021	2.0	Lidia Pivovarova (UH-CS)	Finalizing the text taking into account reviewers' comments
28/04/2021	2.1	Lidia Pivovarova (UH-CS)	Addressing comments of the project lead and advisor
30/04/2021	3.0	Antoine Doucet (ULR)	Minor adjustments and submission
21/12/2021	4.0	Lidia Pivovarova (UH-CS)	Draft update with work done during the project extension
22/12/2021	4.1	Hannu Toivonen (UH-CS)	WP5 leader's check
17/01/2022	5.0	Lidia Pivovarova (UH-CS)	Final update, addressing reviewers' comments
21/01/2022	5.1	Hannu Toivonen (UH-CS)	Final check of the WP5 leader
31/01/2022	6.0	Antoine Doucet (ULR)	Minor adjustments and submission

Executive summary

This document describes the Investigator component of the NewsEye Personal Research Assistant. The Investigator takes as an input a user query that defines a collection of news articles, preprocessed by means of tools developed in WP3 (Text Enrichment). Utilizing tools developed in WP4 (Dynamic Text Analysis), the Investigator analyzes the input collection and finds statistical characteristics potentially interesting for the user.

This document describes the Investigator architecture, the underlying algorithm and implementation details. The main contribution is made in the conceptualization of the Investigator as a self-adaptive exploratory search engine. In the current implementation the Investigator is not only able to produce a non-trivial analysis of the data, but also allows for further extensions and adding more autonomous properties.

Conceptualisation was performed during project Years 1 and 2 in parallel with the implementation, while Year 3 and the project extension period were devoted to finalizing the implementation and integrating the Investigator and the whole Personal Research Assistant into the NewsEye platform.

Contents

Executive Summary	3
1. Introduction	5
1.1. Personal Research Assistant Overview	6
1.2. Personal Research Assistant Architecture	7
2. Investigator Algorithm and Architecture	8
2.1. Investigator as an Exploratory Search Engine	8
2.2. Investigator as a Self-Adaptive System	10
2.3. Update Function	11
2.4. Investigator Strategies and Investigator Decisions	12
2.5. Task Selection and Execution	13
3. Data Aggregation and Visualization	13
3.1. Data Aggregation within the Personal Research Assistant	13
3.2. Location Visualization Tool	15
4. The Role of the Investigator within the NewsEye Platform	16
4.1. Interaction with Reporter and Explainer	17
4.2. Investigator Database	17
4.3. Integration of various processors into the Assistant	19
4.4. User interaction via the Demonstrator user interface	20
5. Implementation Notes	22
5.1. Personal Research Assistant API	22
5.2. Interaction with Demonstrator	22
5.3. Task Queue	23
5.3.1. Asyncio Library	24
5.3.2. Celery Task Queue	24
5.3.3. Comparison	25
5.4. Code Reusability	25
5.5. Processor Inventory	26
6. Conclusion	27
A. API specification	31
B. SpaceWars: A Web Interface for Exploring the Spatio-temporal Dimensions of WWI Newspaper Reporting	33

1. Introduction

This document describes the Investigator component of the NewsEye Personal Research Assistant (PRA), able to analyse large collections of historical news using an extensible inventory of text-processing tools, developed in work packages 3 and 4.

The Investigator performs exploratory analysis of a news-article collection on behalf of the user to discover potentially interesting phenomena. The Investigator acts within the modern exploratory search paradigm [1, 2], though it uses a broad inventory of text processing tools that can be applied to various document sets depending on the user query. Consequently, it requires complex action planning. We are unaware of any similar complex exploratory analytic system for humanities.

Intelligent personal assistants have been employed in various applications, due to their ability to provide context-based support to users efficiently, saving time and allowing them to focus on important tasks: e.g., navigation [3], autonomous vehicles [4], collaborative software requirements engineering [5], time management [6], assisting bat researchers [7], e-mail organization [8], or patient healthcare [9]. The degree of sophistication and autonomy of such artificial assistants varies depending on the application and type of the required assistance, from performing one-off tasks (e.g., setting up a reminder) to applying computational models (e.g., classifications and information extraction) and autonomously searching for solutions and optimizing situations (e.g., schedule management).

Scholars have special information needs and require support for corpus management [10]. Historians are typically interested in analyzing historical data on a level of abstraction that computational models cannot fully learn on their own, e.g., seeking answers to ‘why’ and ‘how’ questions. Nonetheless, computational and statistical methods can aid them in analyzing and studying massive collections of historical newspapers, for instance, to highlight usages of certain concepts over decades [11], to investigate changes [12] or to detect neologisms [13]. Applying potentially informative computational analyses on multiple sub-collections is not only tedious and time-consuming, but sometimes ruled out by the lack of easy-to-use tools and specialist skills, such as programming. As a result, new tools are needed that are capable of automatically analyzing historical data while giving historians the freedom to dynamically adjust the parameters and context of the analysis.

The NewsEye Assistant aims at developing novel methods facilitating access to digitized historical newspapers for a broad range of users, including professional historians as well as the general public. Users interact with the Assistant through a web-interface, where the Assistant returns the results of the Investigator’s autonomous search, along with automatically generated natural language reports, when applicable. Though the NewsEye is focused on historical research and deals with historical newspapers, we believe the same tools and design principles are applicable in other humanities disciplines, where objectivity is a crucial issue.

This report is organised as follows. This section overviews the Personal Research Assistant architecture and its role within the NewsEye project. In Section 2 we describe the internal architecture of the Investigator, its core algorithm, with special focus on decisions made to continue investigation based on the previous results.

In Section 3 we explain how results of analysis are aggregated, summarized and visualized within the Investigator. Part of this work has been presented at the Histoinformatics workshop at the JCDL conference [14] and is attached as an appendix to this report.

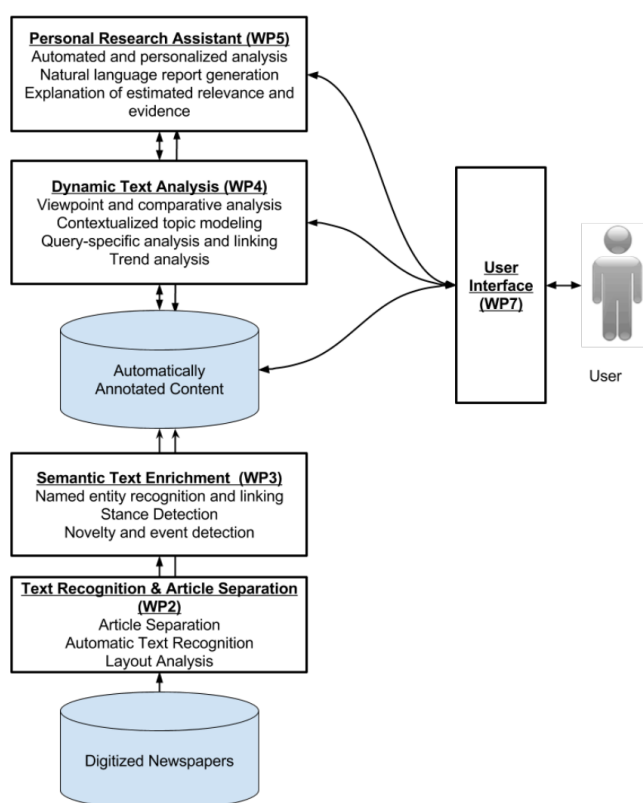


Figure 1: An overview of the NewsEye concept. This document describes some functions of the Personal Research Assistant (work package 5).

In Section 4 we overview the Investigator's role within the NewsEye data analysis platform and its interactions with other parts of the project.

In Section 5 we overview technical details on the implementation of the Investigator and the Assistant in general. This information is crucial for the project sustainability plan and potential deployment of the Personal Research Assistant for practical use.

We conclude in Section 6 by briefly overviewing work done in the respective task of the NewsEye project (Task 5.1) and evaluating its compliance with the project plan.

1.1. Personal Research Assistant Overview

The general information flow within the NewsEye infrastructure is presented in Figure 1. Images of scanned newspapers are provided by the National Libraries of Austria, France and Finland. The images are processed to extract text and separate pages into articles (work package 2). Articles are then semantically annotated by a number of NLP methods including named entity recognition, sentiment analysis, and novelty and event detection (work package 3). All these operations are performed offline and the results are stored in the database. Dynamic text analysis is run on demand and performs query-specific analysis of sets of documents, document linking and comparative analysis of multiple document sets (work package 4).

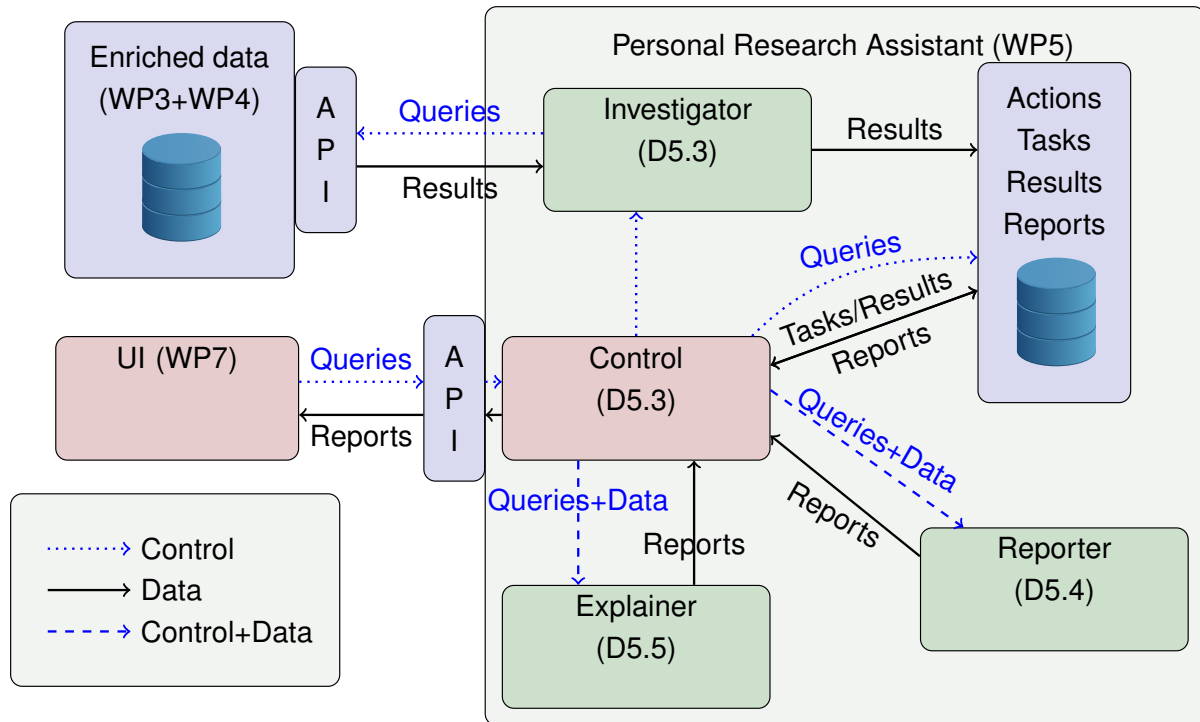


Figure 2: The Personal Research Assistant (‘PRA’, work package 5) with components developed in work packages 3, 4 and 7 shown to demonstrate how the Assistant interfaces with those.

The Personal Research Assistant deals with heterogeneous data and a variety of analytical tools. The goal of the Assistant is to make effective use of these tools to find peculiarities of potential interest for humanities research (Investigator). The Assistant produces a set of natural language reports detailing its findings (Reporter) and detailed explanations of the analysis steps that led to the findings (Explainer).

Users interact with the Assistant via the NewsEye Demonstrator (developed in work package 7). The user interface allows users to query data on various levels. First, it is possible to directly query the database index to simply collect data. User can save and combine the search outputs to build a dataset. Then the Investigator starts autonomous exploratory analysis based on that dataset. The autonomous operation of the Assistant complements the classical user-operated search features of the Demonstrator. The desire to include autonomous operations stems from humanities studies where the option to approach history without predefined questions is seen as a key advantage of modern data-driven methods.

One of the key properties of the interface is that it allows the user to repeat all steps performed by the Investigator, see their results and change parameters of the analysis utilities. In addition, the user can directly apply any analysis tools to a collection. The results obtained using this manual analysis are presented in the interface in the exact same form as results of the autonomous investigations. To facilitate this, the Assistant API contains additional entry points that wrap calls to dynamic text analysis (WP4), as will be described in more details in Section 5.1.

1.2. Personal Research Assistant Architecture

An overview of the Personal Research Assistant internal architecture is presented in Figure 2. The Assistant consists of three primary components—Investigator, Reporter and Explainer—and a Controller

component. The Controller component has two primary functions. First, it provides an Application Programming Interface (API) for users, especially the NewsEye User Interface (UI, see Deliverable 7.8). This allows users to view the Assistant as a single, unified system so that they do not need to concern themselves with the internal division of labor within the Assistant. The API is used via HTTPS queries and is described in more detail in Section 5.1. Second, as the name implies, the Controller provides a central control mechanism that passes messages and results between the three major subsystems of the Assistant, i.e. the Investigator, the Reporter and the Explainer.

This design facilitates the distribution of the Assistant's components over multiple virtual or physical machines if such a distribution would become needed due to increasing amount of users. Modifying the Assistant so that a single Controller instance acts as a load balancer and distributor of work to multiple instances of the subcomponents, while not truly trivial, would be relatively simple following standard approaches used in many other online services. However, in the current implementation we focused on the core Investigator algorithm and minimized efforts needed to deploy and maintain the system.

The Investigator component, in broad terms, autonomously performs a series of queries over a newspaper corpus using different tools provided by work packages 3 (semantic text enrichment) and 4 (dynamic text analysis) to identify potentially interesting factors from the data, as described in more details further in this deliverable.

Having received the analytical results from the Controller, the Reporter then transforms the results into natural language expressions. This transformation process is discussed in significant detail in Deliverable 5.7. The resulting natural language document is then returned to the Controller. The Controller then sends the document to the party that requested it. In the most likely scenario, this is the NewsEye User Interface (Demonstrator) that will display it to the end user. In addition to this, the Assistant contains a database component which stores analysis results obtained from the Investigator, reports obtained from the Reporter and other necessary data. All results of analysis and intermediate steps of Investigator are recorded in the database, which allows asynchronous function calls as described in Section 2. This internal database is different from the main Solr index, which contains all datasets, annotations and metadata.

Finally, the Assistant contains an Explainer component. Whereas the Reporter reports *what* the Investigator found in the corpus, the Explainer explains *how* those findings were obtained and *why* the Investigator believes them to be of interest. In other words, whereas the Reporter describes the end result of a process, the Explainer describes the process itself. The Explainer component is described in the public Deliverable 5.8. The requirement of interpretability (explainability) of the Investigator's actions impose additional constraints on the Investigator. First, it has to record all its actions in a form that later can be converted into a natural language explanation. Second, it avoids some non-trivial sequences of analysis if they cannot be clearly explained. This topic is addressed in more detail in Section 2.

2. Investigator Algorithm and Architecture

2.1. Investigator as an Exploratory Search Engine

The Personal Research Assistant, and in particular the Investigator component within it, performs exploratory corpus analysis on behalf of the user. The intent is not to replace the user's own exploration, but rather to provide complementary automated utilities to help the user discover interesting new phe-

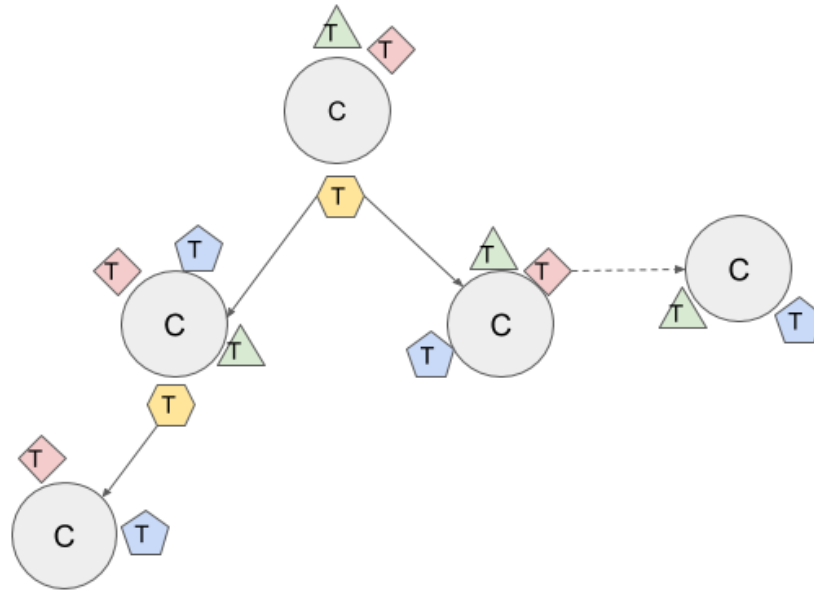


Figure 3: Schema of the text collections exploration process. **C** means a collection of articles, **T** a single analysis task performed on this collection; a **solid arrow** denotes that a collection is a subset of the previous collection, a **dashed arrow** means that a collection is related to a previous collection.

nomena in the newspaper data. According to White [2], ‘*exploratory search describes an information-seeking problem context that is open-ended, persistent, and multifaceted, and information-seeking processes that are opportunistic, iterative, and multitactical.*’ This is a close match to our view of the desired functionality of the Investigator and the Personal Research Assistant in general.

The user starts interactions with the Assistant with a general question “tell me something (statistically) interesting about article dataset X”. Here dataset X could be, for example, defined by the query “articles in Arbeiter Zeitung between 1913 and 1920”, or it could be a manually curated dataset already collected by the user. The Investigator then tries to identify interesting patterns, subgroups, or trends that relate statistically to the contents and metadata of the given news articles, via term or collocation frequencies, named entities, topics and other analytical results in the subcollections.

Autonomous investigations are implemented as navigation among various sub-collections. The investigator finds subcorpora that are potentially relevant to the users’ research question, and also splits the collection—most importantly by date, but also by the newspaper title, by location, or by usage of certain keywords—and then compares those parts.

The exploration process is schematized in Figure 3. In the schema, article *collections* are shown as gray circles labeled with **C**. A root collection is defined by the user and is either a *search query* or a manually compiled *dataset*. A collection is analysed by a number of *processors* implemented as parts of WP4 (Dynamic Text Analysis), for example, keyword extraction or building of timeseries. The ‘processing atoms’ of the investigator are *tasks*, which are shown in Figure 3 as polygons labeled with **T**.

Processors could be classified by the type of procedure they are doing: analysis of an input collection, comparison of two or more collections, manipulation of an input collection, e.g., clustering, outlier detection, finding related articles. Respectively, tasks have different types depending on the underlying processor. *Analysis tasks* produce the main results in the Investigator’s findings. Other types of tasks are *dataset manipulations*—e.g., dataset splits, or finding related data collections—and *dataset comparisons*. Some processors take articles as input, while others work on results of the previous tasks.

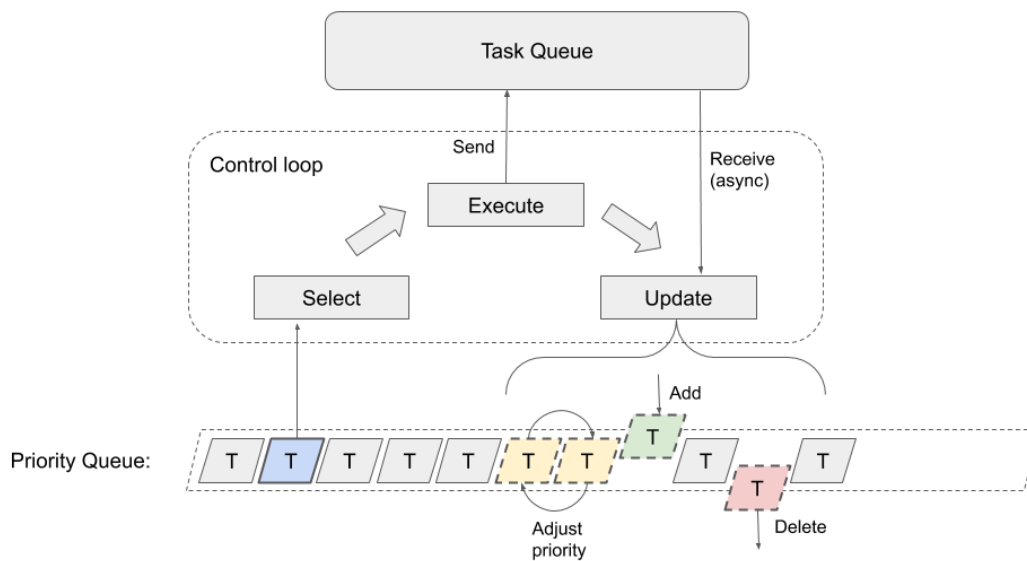


Figure 4: Task queue.

Thus, tasks are organized into *paths* that contains actions that logically follow one another.

The Investigator makes a decision to proceed on new collections based on the results of the tasks applied on the current collection. These new (sub)collections are created by some of the tasks, identified as the origins of arrows in Figure 3. The new collections typically are subcollections of the previous one. However, the Investigator could also explore a new set of articles somehow related to the current one, e.g., by a similar topic distribution. This kind of relations is denoted in the schema with a dashed arrow.

To sum up, the Investigator’s computational graph is organized as a tree, where each node is a collection of articles with a set of applied tasks. Relations between nodes are defined mostly by query transformations and rarely by other relations between articles. This graph is kept in the Investigator’s internal database and plays a central role for the following parts of the algorithm:

- keeping track of the status and relations of tasks,
- planing, i.e. reasoning about the future tasks to take,
- evaluating interestingness of the analysis results,
- keeping information about the “critical computation path” for the Explainer.

2.2. Investigator as a Self-Adaptive System

The high level software architecture of the Investigator is that of a *self-adaptive system* or a *control process*, which is illustrated in Figure 4. The *controlled element* is the exploratory search described in the previous section. The control of the search process is realised through maintaining a *priority queue* which contains tasks to be *executed*. The manager element *updates* the priority queue and *selects* the next tasks for execution.

The task execution and the updates take place *asynchronously*; the manager element may choose to update the priority queue whenever it receives new results from the executed tasks, e.g., by adding new tasks or adjusting the priority of the currently pending tasks. The new tasks are selected to be

executed periodically by polling the *external task queue*, which executes the tasks (and may be shared with the other parts of the Assistant), for available resources. The external task queue implementation is covered in Section 5.3.

The above elements from the Investigator may be fitted to the general self-adaptive software architecture called MAPE-K [15]. Although self-adaptive systems typically consists of a base system, which is a piece of software, and a manager which controls the software, similar parlance may be used in the Investigator where the base system (i.e. the controlled element) is a more abstract search process and the manager controls how the search process unfolds through a *feedback loop* from the search process back to the manager element. The feedback loop in the Investigator is realised through the received results from the executed tasks and managing the used resources, e.g., by monitoring the execution times of the tasks.

In MAPE-K, the manager's control over the base system is divided into four conceptually separate, consecutive steps: monitor (**M**), analyse (**A**), plan (**P**) and execute (**E**), where all of these steps may use a shared knowledge base (**K**). Monitoring of the base system is handled through probes or hooks: the manager element either continuously observes the base system or is notified of particular events taking place in the base system, e.g., using callbacks from the finished tasks. The raw input (e.g., a task result) from the base system is refined through analysis, which is used as the basis in planning on how the process should be controlled. Execution realises the planned changes to the process.

In the Investigator, the MAPE-K loop is present in the following way:

- M** Monitoring occurs through asynchronously receiving results from the finished tasks and keeping track of the task execution times (the right hand side arrow from the Task Queue to Update in Figure 4).
- A** The overall analysis of the results is handled by the Update function (on the right hand side of the Control loop in Figure 4). The Update function may further delegate the analysis tasks to other functions with more specialized responsibilities.
- P** Planning is done by maintaining the priority queue (the bottom element in Figure 4), which the Update function manipulates.
- E** Execution is periodical by selecting the (set of) tasks with the highest priority and sending them to be executed in the external task queue (the top element in Figure 4), or cancelling currently running tasks because of too long execution times.
- K** All the above elements may alter the Investigator's database which is used as the shared knowledge base (not shown in Figure 4, but cf. the right hand side of the Assistant in Figure 2).

2.3. Update Function

Update function is a core component of the Investigator, which ensures a non-trivial adaptive behaviour. Decision on what processors to run next are done based on the latest completed actions. Since many tasks are executed in parallel within one investigation run, at each step the Investigator deals with several collections: the initial collection specified by the user, results of various splits, related documents. These collections could be at various stages of processing, i.e. just created, analysed with certain tools, compared to other collections and so on. From an implementation point of view, it is easier to keep those

steps separated, rather than force the Investigator to make decisions based on the heterogeneous set of all executed tasks.

Thus, the Investigator *paths*, i.e. sequences of logically connected steps, are stored explicitly. Each step involves several processors and a set of related collections, e.g. results of a split. Then *Update* procedure is organized as a loop through all non-finished paths that updates each of them independently. Decision is done by looking on the current collections and the last action made in this path.

To simplify the implementation, processors are organized in *processorset*, i.e. a set of processors that has a name and that the Investigator could call in parallel. For example, DESCRIPTION processor set includes extraction from collections of the most prominent keywords, names, topics and facets; each of these aspects is implemented in a separate processor but they are all called in parallel for each new collection or subcollection.

Implicit implementation of paths and processorsets simplifies decisions that Investigator is doing repeatedly, such as ‘what to do if the last step concerns one collection, where all documents are in the same language and the last executed processorset is DESCRIPTION?’ or ‘what to do if the last step concerns several collections in different languages and the last processorset is EXTRACT NAMES?’.

2.4. Investigator Strategies and Investigator Decisions

Strategies are implicit properties of an Investigator run. Currently, two strategies are available:

- *elaboration*, i.e. finding potentially interesting facts about a user collection;
- *expansion*, i.e. finding more documents potentially relevant for the user needs.

Strategies are implemented on two levels:

- Global strategies, which are set by a user in the beginning of the run and never change;
- Local strategies, which are properties of a particular path and could switch several times during processing.

Investigator decisions take into consideration both local and global strategies. For example, if the current path strategy is *expansion* and the last processorset is EXPAND QUERY, meaning that some new collection is already found, then the decision is done based on global strategy. If the global strategy is *elaboration* then Investigator continues to analyse the new collection, which could potentially lead to more expansion queries in the future. However, if the global strategy is *expansion* then the analysis is stopped at this point, since digging deeper in the new collection does not comply with the user initial goal.

Potentially, it is possible to introduce more strategies, which would lead to more fine-grained variations in data processing.

To sum up, the Investigator makes decision on how to proceed investigation in a particular path based on the following aspects:

- Global strategy
- Local strategy

- Last processor set
- Collections size and languages
- Interestingness of the previous results

Currently decisions are implemented using rule-based approach, with hard-coded conditions on all those aspects. We believe, that a PRA intelligence could be potentially increased on the decision level, without changing the general Investigator algorithm (and code). Even with the rule-based approach the Investigator is able to demonstrate a non-trivial behaviour: the set of performed tasks significantly varies depending on the input data.

2.5. Task Selection and Execution

Since PRA runs many tasks in parallel and some of them require heavy computations, execution needs to be prioritized. The Investigator returns processing results in portions, as soon as they were obtained. Thus, the order of task execution affects the user experience.

When a new task is added into priority queue, the task priority is specified as a combination of the following aspects:

- Pre-defined processor priority, which is specified based on computational resources that a processor needs, the load it would impose on the Solr index and whether it is used for providing data for the downstream tasks;
- Context priority assigned by Investigator depending on the current situation;
- Interestingness of the previous results produced in this path;
- The number of times the processor was called in this run—this option ensures that the user would see diverse results as early as possible.

In the current implementation, all planned tasks are executed even though some of them could stay in a waiting list much longer than others.

3. Data Aggregation and Visualization

Apart from the development of the Investigator itself, work in Work Package 5 included implementation of integrative views on results of other work packages that summarize and visualize results of data analysis. As explained in Section 2.1, the Investigator works with *collections* of texts, while many text analysis tools developed in Work Packages 3 and 4 process each *document individually*. Thus, the Investigator needs to perform an additional aggregation step to obtain a summary view on the data, both to continue analysis and to give meaningful representations to the user.

3.1. Data Aggregation within the Personal Research Assistant

The Investigator provides aggregated views for several tools of the tool set. For some of them, there is a standard way to aggregate data. For example, to summarize the vocabulary of a collection, we use the well-known tf-idf score. In some other cases, aggregated views are readily provided by other Work Packages that produced analysis tools. This is the case for topic modelling (see Deliverable 4.7 "Intelligible representation of statistical analyses"). We developed in Work Package 5 an aggregation technique for named entity recognition (Deliverable 3.5) and stance detection (Deliverable 3.6).

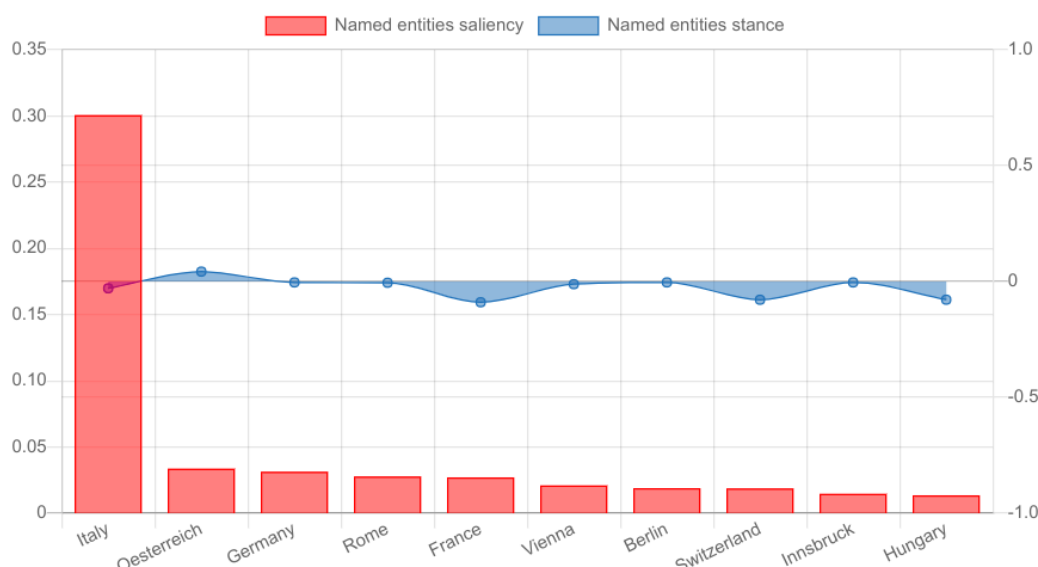


Figure 5: A screenshot from the NewsEye platform: the most salient place names for a user collection.

The named entity recognition (NER) tool only locates names in individual documents, and a separate aggregation step is needed to identify names that are likely to be important in the document collection. A partial solution is that low-frequency names can be ignored with relative safety: rare names are less likely to be important for the collection. OCR errors in historical news texts also produce spurious names with low frequencies.

In order to rank high-frequency names in an approximate order of importance or salience, we take into account both the frequency and the prominence of names in the individual articles as follows. We compute a measure of *salience* first for each entity in each document of the collection. Salience is computed as a geometric mean of prominence and frequency of the name within a document [16]; the most salient names are those that are mentioned closer to the beginning of the document (i.e., more prominently) and repeated several times. The saliences are then averaged for each entity across the whole collection.

As a result of this procedure, both spurious and less important names tend to get lower salience and can be excluded from results. The most salient entities are used for further analysis: for example, in the current implementation, the Investigator uses the most salient names to query the stance detection tool. Figure 5 shows a resulting diagram with the most salient place names and their stances in a dataset. Such diagrams, as well as verbal descriptions of the salience and stance analysis and aggregation, are reported to the user via the Reporter.

Stance detection tends to output neutral stance much more often than negative or positive. This is due to the fact that newspapers generally present information in a neutral tone, but also due to difficulties of stance detection in historical documents (see Deliverable 3.6). Additionally, positive and negative stances cancel each other out when taking an average over documents. These effects can be seen in Figure 5, where stance values often are neutral (zero) or close to neutral.

To provide a potentially more useful aggregated viewpoint to stance, we also produce a temporal plot for each entity where negative, positive and neutral mentions are aggregated separately. In Figure 6,

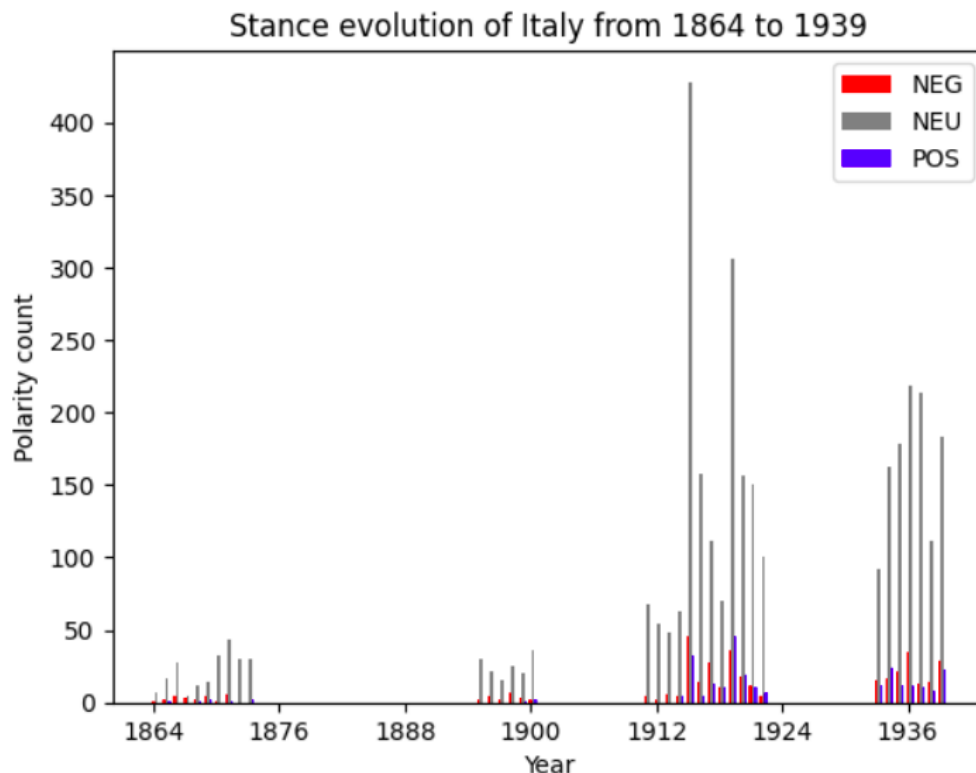


Figure 6: A screenshot from the NewsEye platform: stance evolution for an example name from a user collection.

all mentions of Italy in the user's data collection are grouped by year; for each year, we then show the numbers of positive, negative and neutral mentions. By aggregating positive and negative stances separately we can show more informative dynamics of stance evolution than with a averaging strategy.

3.2. Location Visualization Tool

We developed a location visualization tool [14] for historians, in order to aggregate and show distributions of location names in a user-friendly manner. The tool shows locations extracted from the NewsEye collection on a geographical map and also provides various tools to analyse locations in context. Visualizing locations from a text collection in a geographical information system is a digital media transformation that puts the focus on the spatial aspects of the data.

We illustrate the location visualization tool with the period of the First World War. The NewsEye collection contains newspapers from Austria, France and Finland, which belonged to different parties during WWI and got their news from different channels. We use a subset published between 1913 and 1920. It covers five different newspapers from three different European languages: *Arbeiter Zeitung* and *Illustrierte Kronenzeitung* (German), *Le Matin* and *L'Œuvre* (French), and *Helsingin Sanomat* (Finnish). The locations have been extracted and tagged by tools from Work Package 3 (Deliverable 3.5).

Our tool enables users to explore the perception and influence of space and place in relation to other dimensions of the Great War¹. For this, the tool consists of two interfaces: (1) a map navigation module, shown in Figure 7, that presents extracted locations on a map, and (2) a concordancer that presents

¹ A web interface to the WWI location data is available at <http://spacewars.newseye.eu/>

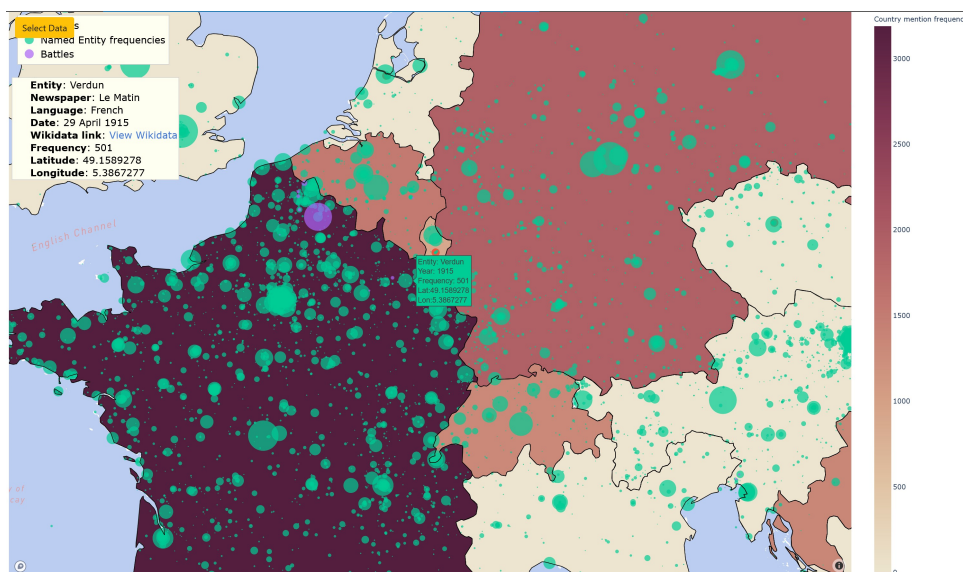


Figure 7: Visualizing every entity in *Le Matin* (1913-1915) through the map module

words most often collocated with a given location. The concordancer is linked to the articles where the name was found, allowing switching between distant and close reading.

A paper describing the location visualization tool [14] was presented at the Histoinformatics workshop at the JCDL conference and is attached to this report as Appendix B. The corresponding code is publicly available at <https://github.com/dhh21/SpaceWars>.

4. The Role of the Investigator within the NewsEye Platform

In this section we describe how the Investigator is related to other Assistant modules, user interface and other work packages within the NewsEye project. The interactions between the Investigator, the Reporter and the Explainer are coordinated by the Controller module and supported by the internal database, which stores all Investigator actions as well as individual tasks, results and reports. Controller also handles the API call for the user interface. From the interface point of view, the Assistant is a solid unit.

The input data for the Investigator are prepared in work packages 2 (text recognition and article separation), 3 (semantic text enrichment) and 4 (dynamic text analysis). The textual data, enriched with named entities and stances, are stored in the NewsEye database and made accessible through the Solr index. Text analysis tools, developed in work package 4, work *by request*; they comprise a *processors inventory* for the Investigator. They get an input definition and task parameters from the Assistant and return results, which are stored in the internal database and returned to the user in a form of raw outputs or natural-language reports.

The idea is to consider the user and the Assistant equal in many respects. The user has access to the operations that the Assistant has access to, especially processors. This allows the user to replicate, experiment with, understand and improve the actions of the Investigator. However, the user has access to functionalities that the Assistant cannot use, e.g., a user can compile a dataset by hand-picked articles, while the Investigator operates only on a query level.

Accordingly, the user interface of the NewsEye Demonstrator provides two modes to access the functionalities of the Investigator: *autonomous mode*, where the user starts the Investigator and gets the most promising subset of its results, and *manual mode*, where the user processes a certain text collection with a particular tool and manually specified parameters. In the autonomous mode, the user specifies a data collection that needs to be analysed and sets the Investigator strategy—either elaboration or expansion on the similar datasets. The task-running mode is out of the scope of work package 5 but it is taken into account when developing the Investigator. It is crucial that the user can manually reproduce Investigator's actions, when necessary changing the tool parameters.

We now discuss relations between various parts of the platform in more details.

4.1. Interaction with Reporter and Explainer

The detailed description of the Reporter internal structure and algorithms can be found in Deliverable 5.7. The interaction between the Reporter and the Investigator is driven by the Controller. The Investigator runs (sequentially or in parallel) a number of analysis tasks. In the end of each execution step the Investigator scores each task with its potential interestingness for the user. The Controller then sends a request to the Reporter component for producing a natural-language report based on the result obtained by the Investigator.

The input of the Reporter is a list of scored task results. A result is scored as a whole and in addition each unit within the result has its own score: e.g. if a task produces a keyword list, then the result contains an overall interestingness and each keyword has its own interestingness. The Reporter selects a set of facts to talk about based on the interestingness scores assigned by the Investigator, produces the report and returns it to the Controller, which adds it to the Assistant's database, from where it can be retrieved and returned to the user.

The Explainer (Deliverable 5.8) also produces natural-language texts based on the Investigator actions. The main difference is that if Reporter describes *what* results are obtained by the Investigator, the Explainer explains *how* these results were obtained and *why* the Investigator run these tasks in that order. In other words, Explainer provides user with more information on edges in the exploration tree shown in Figure 3.

Implicit investigation paths simplify coordination with the Explainer. When the Controller receives a request for an explanation for a certain task, it identifies the path containing the task and sends to the Explainer all steps between the root action and the target task. Each Investigator decision is accompanied by *motivation*, i.e. a short message from a predefined dictionary. Motivations are obtained in the *Update* step and stored within the paths. Thus, no additional reasoning is required to trace a sequence of events that led to this task.

4.2. Investigator Database

The Internal database of the Investigator allows it to work asynchronously as it stores all steps and all results of the exploration process. The database schema is shown in Figure 8. We now briefly describe each table in the database.

Investigator Run stores the basic information of a single investigation process. Each run relates to many **Investigator Actions**. Actions are large-scale steps of the analysis, which could have one of

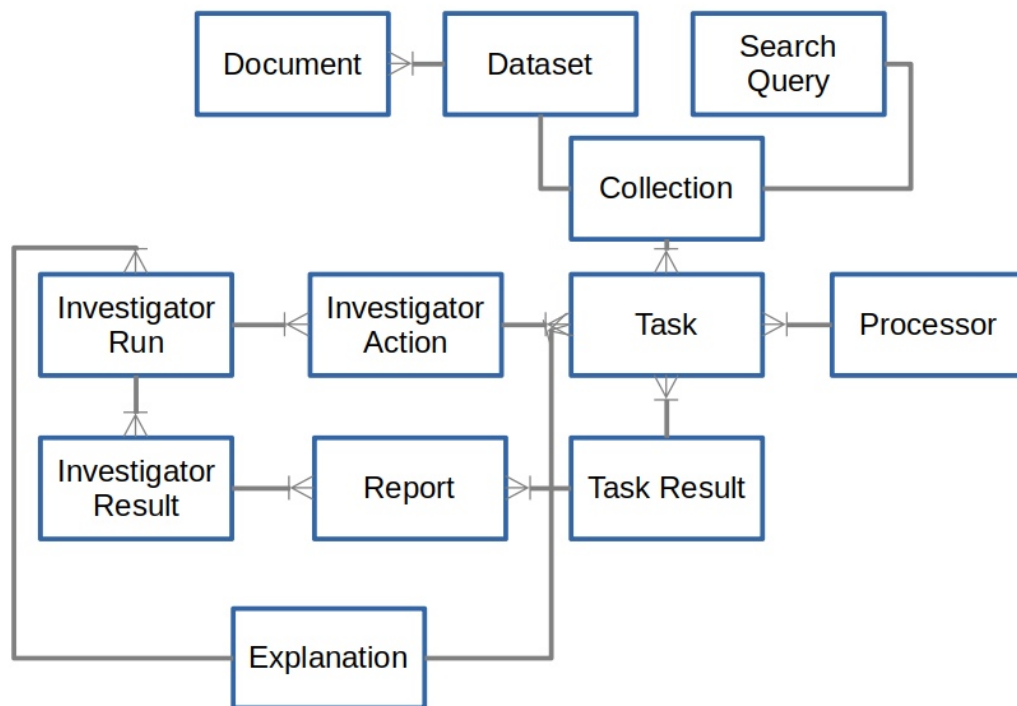


Figure 8: Internal Database of the Investigator that enables asynchronous analysis of the data.

the following types: *initialize* task queue, *select* tasks from the queue, *execute* selected tasks, *update* task queue, or *stop*. Among other attributes, the database table for actions stores the input and output states of the task queue, the list of associated tasks and the sequential number of the action within the run—these attributes allow reconstructing the execution path that led to a certain task, and to continue the process from a given step. The action table also stores the *motivation* for selecting or executing the associated tasks.

Tasks are the main execution atoms, as was explained in Section 2. Consequently, the Task table is related to many other tables in the Investigator database, as could be seen in Figure 8. A task is a certain processor applied to a certain document collection, thus it has relations with **Processor** and **Collection** tables. A collection could be either a query or a dataset. **Dataset** table also has relation with document table to store document id and relevance. **Search Query** table does not need this relation since documents could be retrieved from the database on the fly.

The relation between actions and tasks is one to many, meaning that an action could be associated with many tasks but each task belongs to exactly one action. In case the Investigator or the user decides to run a task with exact same parameters as previously existed task, a new record is added into Task table. Thus, each record is associated with a particular call of a processor, with a unique id and a unique position within an Investigator run. A task record contains only collection id, processor id and call parameters, so redundant records do not take much space. **Task Result** records, on the contrary, contains actual result produced by a processor, which could be quite long, especially if the underlying collection is big. Thus, result records are not copied. If an identical task has been executed before, the corresponding result id is added as a new task record, so, as shown in the figure, the relation between tasks and results is one to many: each task has exactly one result but result may belong to many (identical) tasks.

Tasks called directly by the user in the manual analysis mode are stored in the same format in Task table.

Investigator Result is obtained by a set of tasks. Each Investigator Run is associated with multiple results, since they are stored at various steps of the analysis, so that the user can see intermediate results without waiting for the final ones. As could be seen in the figure, both Task Result and Investigator Result table are related to **Report** table, which is used to cache the Reporter output. Each result may have more than one report, for example, written in different languages.

Similarly to reports, explanations are stored in **Explanation** table. However, explanations do not depend on the results and are connected directly with runs and tasks. Paths are currently stored in a JSON object as one of the **Investigator Run** properties, thus these are the only tables needed to produce an explanation.

4.3. Integration of various processors into the Assistant

The Investigator is a meta-analysis tool, which utilizes text processors developed in other work packages. In order for the generic Investigator to function it utilizes information on the data structures and applicability of the processors at each stage of the analysis, and estimates the interestingness of the outcome. The decisions are done based on the properties of the current collection and applicability of the tools for such collections (e.g. certain tools are not implemented for certain languages).

Outputs of WP3 (Text Enrichment), namely named entities, stances and events extracted from news articles, are stored in the NewsEye database, from which the Investigator directly queries them together with the article texts obtained from WP2 (Text Recognition and Article Separation). Availability of the WP3 outputs for all or majority of the data is crucial for Investigator to make various views on the same datasets: e.g., a topic distribution and a list of the most prominent names gives different views on a data collection and the Investigator can choose between them to return the most interesting aspect to the user.

WP4 (Dynamic Text Analysis) makes text analysis *on demand*, i.e. in the terminology of this deliverable, it provides Investigator with *analysis processors*. To use a processor, the Investigator should know what are its input and output data and how to estimate the interestingness of the results. Thus, each processor specifies its input and output types and is accompanied by a separate function that estimates interestingness. Each processor outputs a *dictionary*, i.e. a list of key-value pairs, where keys and values have certain types, defined in advance. Key types and value types could be interpreted as *semantic type* and *mathematical type* of the data, respectively.

(Semantic) key types supported by the system are:

- article ids
- strings: words, lemmas, named entities
- collocations
- stances
- topics
- facets: i.e. metadata, such as article sources, languages, dates, etc.

(Mathematical) value types supported by the system are:

- numbers
- vectors
- probability distributions
- sequences
- sets of (nominal) features

For example, topic modelling outputs a dictionary where keys are article ids and values are topic probabilities; name entities are represented with a dictionary where keys are entity ids and values are their salience in a given collection.

The technical details on how to add a new tool into the Investigator's inventory is described in Section 5.5, together with the list of available processors.

4.4. User interaction via the Demonstrator user interface

The user interface developed in WP7 (Demonstration and Exploitation), as part of the NewsEye Demonstrator, is described in detail in Deliverable D7.8 (Demonstrator). In this section we only touch on the requirements to the interface imposed by the Investigator. The Investigator acts on behalf of the user and the actions are considered equal to those of the user. Technically speaking, the state of any dataset and result does not differ based on whether the state was reached by the user or by the Investigator. This simplifies the user's interactions with the Assistant and ensures conceptual simplicity and clarity.

The system maintains the specification of steps taken to reach any particular state, whether performed within an Investigator's run or directly by the user. Technically, it means that the user interface calls the processing tools using the same interfaces as the Assistant. Actual search queries are stored where possible rather than just resulting article sets, in order to support viewing, replicating and modifying the exploration steps.

Storing all user actions allows for reconstructing of a user-defined computational graph similar to the tree shown in Figure 3. The main difference would be that the user tree would have more nodes that have no clear relation to the previous steps—the user might switch between different data collections based solely on research intuition, while the Investigator needs a particular reason to move from one collection to another.

The Demonstrator allows the user to navigate the computational graph, i.e. to show a current state of the analysis and results obtained at each step.

Implementation of the interface is outside our scope in work package 5. However, we formulated a list of functionalities that a Personal Research Assistant should have in a complete implementation:

1. Basic Functionality

- to deal with articles and article collections and anything directly related to them, as well as with queries; to be able to start any tool that takes articles as input, i.e. related article finders, collection analysers and collection comparators.
- to handle articles and article collections produced by any tool, i.e. related article finders or dataset splitters
- to keep track of which tool produced any article collection and with which parameters

- to allow for saving and naming article collections, and loading of previously saved collections for further processing
- to be able to initiate a new Assistant task from any article collection and handle article collections produced by it
- to be able to invoke the Assistant to describe the article collection and to report or explain the process that led to its creation

2. Analysis functionality:

- to handle all analysis—i.e. analysis processors, comparison of collections, finding of related collections— and analysis results.
- to have generic components that allow the user to choose among compatible options—based on the processor inventory and data types
- to use tool- and data-specific components to deal with specific analysis, e.g., to show analysis results
- to be able to embed more prominent components for key analysis types, e.g., interactive visualization for topic models, which are implemented as a part of WP4 (Dynamic Text Analysis) and described in Deliverable D4.7
- to allow for saving and naming analysis results, and loading of previously saved results for further processing
- to initiate a new Investigator task from any analysis result and handle analysis results produced by it
- to be able to invoke Assistant to report the analysis result or explain the process that lead to this result

3. Assistant functionality:

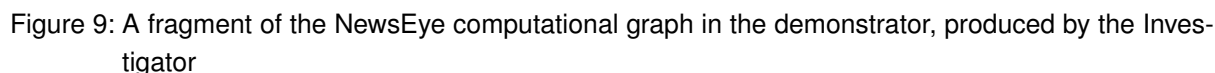
- to manage tasks given to Assistant: start tasks, see running tasks, control tasks, access results of tasks
- when the user accesses results of the Investigator, they are shown in the respective UI component, as if they were produced by the user
- to allow user to save article collections or analysis results

The user interface was implemented in WP7 (Demonstration and Exploitation), with close collaboration between other work packages to take into account the aforementioned functionalities. Its current functionality is briefly described in section 5.2.

The computational graph, produced by the Investigator, is directly presented in the interface, as it is illustrated Figure 9. In the graph, the blue rectangles correspond to article collections, light green rectangles are tasks, where a rhombus is a special type of the task, which splits the collection. Lines represent dependencies between graph nodes. Some tasks have two parent collections—these are comparisons. User could click any node in the graph and get more information on tasks and results. New collections could be saved and reused in further analysis.

More details on the Investigator representation in the user interface could be found in Deliverable 7.8 (Demonstrator), Section 4.3, and also in a PRA tutorial on the NewsEye Youtube channel ².

²<https://youtu.be/YpraHbsiqcM>



The Investigator is implemented as a Python3 package, which uses a Flask REST API engine and a PostgreSQL database to store all internal data, including processor specifications, investigator actions, and article collections. The package ensures the functionality described above but also includes modules that conceptually belong to other Assistant components (Controller, Reporter or Explainer) as well as WP4 (analysis tools). In this section we focus on the components that are crucial for understanding the work from technical point of view.

The Demonstrator accesses the Assistant via REST API, which allows to start data exploration and obtain its results (the scope of this deliverable), as well as to query reports and explanations of the exploration process (Tasks 5.2 and 5.3 respectively). This API, however, has a broader functionality than an access to the Assistant. As we discussed in Section 4.4, the Demonstrator allows to work interchangeably in the autonomous and manual task running modes. Results and reports of a given processor look identical regardless on whether they were initiated by the user or by the Investigator. In practice this means that all processor calls are mediated by the Assistant, which ensures that they output results and store internal information in the same form as the Investigator. Thus, in addition to the Assistant functions, the API wraps calls of text analysis tool which conceptually belong to WP4 (Dynamic Text Analysis).

The API specification is presented in Appendix A. For the sake of completeness—the API is not described in any other deliverable—we present a full overview of the REST API, rather than picking parts that directly serve the Investigator.

The basic procedure of working with the Assistant API is the following:

1. The Demonstrator initializes a new Investigator run via *investigator* entry point with data definition and gets as an output a run uuid.
2. Using this uuid, the Demonstrator invokes the api for results of the run and gets as an output the run status (finished or running) and a list of finished task uuids. If the Investigator is still running the result could be called again later with a different output.
3. The report on the Investigator results (final or intermediate) could be obtained using *report* entry point. Then the Controller redirects the query to the Reporter API, described in Deliverable 5.7.

4. Output of a single task could be retrieved using *analysis* entry point or via the Reporter.

Alternatively, the Demonstrator can use *analysis* entry point to run individual tasks. In this case it gets as an output a uuid of the task, which then can be used to get the task result or report using exactly same API calls as with the tasks started by the Investigator.

According to the project plan the Investigator should be able to interact with the user in to continue the investigation. In other words, a user should be able to modify a computational graph produced by the Investigator. However, implementation of this functionality would require more time investment into the interface that was feasible within the project. Nevertheless, in the current version of the Demonstrator a user is able to store any collection generated by Investigator and run any processors on these collection, thus manually reproducing and modifying Investigator runs.

The missed functionality can be implemented if the Demonstrator would send to the Investigator computational graphs, computed by users. Then the Investigator would need to parse them to reconstruct analysis paths. Once this is done, the Investigator would continue exploration using the general procedure.

5.3. Task Queue

The *external task queue* used in the Investigator (the top element in Figure 4) may be implemented using multiple technologies. Some of these technologies may not handle the tasks in a queue-like manner, but here we use the term *task queue* to denote all the technologies which may be usable for this purpose. We are currently developing towards two task queue implementations: with Celery, which is a distributed task queue, and with *asyncio* library, which is part of the Python core.

The task queue's main purpose is to handle the asynchronous execution of the tasks the Investigator selects for execution from its internal priority queue. That is, it separates the concern of executing the task from the Investigator itself and thus it provides a proper API for the Investigator to use it. To this end, the task queue implementation has to support at least the following functionalities. There has to be an ability to (1) communicate tasks to be executed to the task queue, (2) get results from the finished tasks (asynchronously) from the tasks queue and (3) poll the status of each task currently in the task queue.

There are multiple ways to implement the functionality described above. For (1) the straightforward solution is to call a function which adds the task into the task queue and the task queue handles the rest. For (2) there are two prominent options, either (a) the Investigator polls periodically the task queue for any finished tasks or (b) each task has a callback to a function in the Investigator which is invoked at the task's completion time. We opt for the option (b). For (3) there are multiple implementation ways, ranging from the Investigator keeping track of the currently running tasks in its own database to asking the task queue directly about their status. As the task queue is seen as an external element to the Investigator, the latter option seems more promising where applicable, as it separates the concern of keeping track of the task status to the external task queue itself. The Investigator may then synchronize its own database with the correct task status when the task gets completed successfully or fails.

Next we give an overview of both of the implementation choices, *asyncio* and *Celery*, with respect to the three points made above.

5.3.1. Asyncio Library

Asyncio is the Python's internal module to write concurrent code via asynchronous event loops, e.g., to distribute tasks via queues. For (1) the implementation is simple, just create a dedicated event loop for the execution tasks and create all tasks into that loop. For (2) the asyncio tasks may be paused using *await* syntax and the execution of the main Investigator's control loop may resume either when the first task has returned or when all the tasks have returned. Another way is to add a callback to the task when the task is done. However, it is suggested to be only used with low-level callback-based code. For (3) the asyncio based implementation offers to return all the currently running tasks, but the status of the tasks, their starting times, etc., are recorded by the Investigator to its database as asyncio offers little help on that side. Thus, most of the bookkeeping related to the task executions is left for the Investigator itself.

The current implementaion allows us to simultaneously run experiments started by several users. However, it is already clear that the implementation fully relying on asyncio will be unreliable in a real setting, e.g. as an electronic library server with hundreds of users. In our experiments, the system becomes unreasonably slow or even crashes when a dozen or more users try to access it at the same time. Thus we consider alternative that could be used if the Personal Research Assistant is deployed in a larger environment.

5.3.2. Celery Task Queue

An alternative for using asyncio could be Celery, a distributed task queue written in Python with a focus on real-time processing. For the Investigator, Celery is particularly promising for (3) because it has its own internal functionality to keep track of the tasks which are currently running and which have already been executed. For example, if the server would crash, Celery typically would be able to begin executing the interrupted tasks as soon as the server is back up and Celery is running again. The Investigator's main database may then be synced with the Celery's knowledge of the task statuses.

A prominent choice to be made when developing with Celery is the choice of the *broker*. The broker acts as the communication route between the worker processes polling for new tasks to execute and the client code which sends new tasks and expects results from the finished tasks. That is, it serves as the means for (1). Typical choices for the broker are *Redis* and *RabbitMQ*, both with their own technicalities which we will not cover here.

Celery allows execution of tasks, subtasks (used to wrap the arguments of a single task execution), callbacks from tasks (with eager or asynchronous execution), task sets (offering a way to execute and return sibling tasks through single functionality), and chords (which executes only after all the tasks in a task set have been executed). This allows to build delicate dependencies between the tasks in the Investigator which are ensured to be executed in the correct order. The Celery may be polled for the statuses of all the currently running or previously executed tasks.

5.3.3. Comparison

Overall, asyncio implementation is easier to develop because Celery adds some complexity, e.g., through the choice and deployment of the broker. However, the advantages of Celery are quite clear. Celery separates the concern of managing the task queue from the Investigator and it allows easily adding new workers in a distributed manner if there is a need to add new resources for processing the tasks. Moreover, Celery's task hierarchy allows to build complex task execution pipelines with assurances of their execution in the correct order. Asyncio, while easily implemented, suffers from the Python's internal one thread per process limitation, and thus each task should be run in its own thread or process to gain concurrency even on a single machine level. This adds complexity to the overall implementation and suffers from the fact that any possible broken tasks (failed for any reason) need to be again initialized by the Investigator.

Current implementation relies on asyncio, however unfolding the PRA into a large-scale system with hundreds of users would require a more sophisticated tool for task control. We envision that the task queue may also be used by the Explainer or the Reporter in the future.

5.4. Code Reusability

The Investigator, as a meta-analysis tool, depends on the processor inventory and the data. In our implementation we have tried to keep the core Investigator's code as general as possible, so that it only keeps track of the priority queue and calls other functions to perform data analysis. This code can be in principle re-used in other projects.

The Investigator currently uses, however, a number of heuristics that rely on the NewsEye infrastructure. For example, an analysis of a given dataset always starts with a fixed set of processing tools which produce an initial description of the dataset. This description includes lists of the most prominent keywords and bigrams for a given collection. This choice has been made to ensure that the description is produced quickly so that the user has some results from the Investigator as soon as possible. However, its implementation was not trivial because for bigger collections extracting word and bigram lists could take a considerable amount of time. Including these processors in the initial analysis set required coordination with WP3 (Text Enrichment) and installation of the Term Vector Component for the Solr database that allows direct access to the database indexes, which already contain the required information and can be queried without expensive string operations. If direct access to the indexes is impossible—e.g., if a project uses a different database framework—the initial description of the data should rely on different set of processors or require some preprocessing done in advance.

The Investigator does not have a user interface of its own and thus depends on the Demonstrator (work package 7). To be able to use the Investigator outside the NewsEye environment it is necessary to adjust the Demonstrator or to develop another user interface. This interface must be able to perform the two primary functions. First, to call the Assistant API, described in Section 5.1 and to convey its outcomes to the user in a meaningful way. Second, to transfer to the Assistant manually-compiled user datasets. Currently, both Demonstrator and Assistant have access to the Solr index that contains the NewsEye news collection. The Demonstrator sends to the Assistant a dataset name, a list of articles within the dataset and manually-assigned relevance for each article. Then the Assistant queries the article texts and their meta-data directly from the Solr index. Alternatively, the user may use the Demonstrator to analyse a collection defined by a search query. In that case the Demonstrator sends this query to the

Investigator, which repeats search to obtain individual articles.

If the Investigator is deployed in some other setting where it does not have an access to the database then a user interface must send a full article text and meta-information, which would require adjusting the processors' code and also taking care of the possible network problems caused by transferring large data collections. The complete code base for the Investigator and Controller is available at <https://github.com/NewsEye/Newseye-WP5-Investigator>.

5.5. Processor Inventory

In addition to the Assistant functionality, the Assistant code contains wrappers for analysis processors. The simpler processors, such as keyword extraction, are fully implemented within the Assistant package while more sophisticated processors, such as topic modelling or text summarization, require calls of the external packages. Regardless of the difference, from the Investigator's point of view all processors have the same structure. All processor are implemented as descendants of the main Processor class, which has three main functions:

- **get_input_data**: specifies queries to Solr or to the external package which needed to obtain the input data; e.g., in the current implementation topic modelling takes as an input articles ids while summarization requires the full text of the articles
- **make_result**: the main analysis function, which takes most of the computational time and outputs the results in a form of json dictionary; data aggregation (see Section 3), if it is required by the nature of the tool, is implemented as a part of this function
- **estimate_interestingness**: a function that works on top of results; in an ideal case it would call one of the predefined interestingness functions that are stored separately; in practice interestingness is mostly defined specifically for a given processor

Whenever a new task is called manually or via the investigator, the data definition is sent to the `get_input_data` function of the corresponding processor, then the output of this function, i.e. data, are sent to the `make_result` function, which output is sent to the `estimate_interestingness` function. The outputs of the latter two functions are stored in the database and can be used by the user via the Demonstrator user interface and the Assistant API.

The following steps are needed to add a new processor into the Investigator's inventory:

1. implement the three aforementioned functions
2. add to the internal Assistant database the processor name and import path, its input and output types and the parameters
3. provide semantic interpretation of the output for the Reporter (Deliverable 5.7)

This is a relatively easy procedure, which has been done several times as we developed the Assistant. Implementation of the processors, however, is not a trivial task, which is more dependent on the particular data at hand than the Investigator itself.

The processor inventory includes the following analysis tools:

1. **ExtractFacets**: Examines the document set given as input, and finds all the different facets for which values have been set in at least some of the documents

2. ExtractWords: Extracts the most prominent words from a given collection
3. ExtractBigrams: Extracts the most prominent bigrams from a given collection
4. ExtractNames: Extracts most salient names from a given collection
5. TrackNameSentiment: Builds sentiment timeseries for most salient names in the collection
6. GenerateTimeSeries: Generates timeseries for a given facet
7. FindBestSplitFromTimeseries: Find the best timeseries from the result of the previous processor and the best (single) split point within that timeseries
8. QueryTopicModel: Queries a selected topic model
9. TopicModelDocumentLinking: Finds similar documents using topic models
10. TopicModelDocsetComparison: Compare datasets using topic models
11. Summarization: Produces a summary of a set of articles
12. Comparison: Special type of the utility which takes as an input a list of tasks with the same input type and finds the difference; it uses a simpler method than TopicModelDocsetComparison but applicable almost any results
13. ExpandQuery: Propose new queries by finding words most similar to keywords in the dataset

6. Conclusion

According to the project plan the Investigator should be able i) to plan, form and run queries using analysis tools developed in work packages 3 (semantic text enrichment) and 4 (dynamic text analysis), ii) to create strategies for investigation, to analyze the results obtained and to adjust its strategy accordingly, and iii) to interact with the user to adjust the investigation plan.

The Investigator is able, after being given some initial guidelines by the user, to iteratively form different search queries, perform various types of analysis to the resulting document sets and use the results from these analysis tasks to guide itself in generating again new searches or choosing different types of analysis to perform on the existing set of documents, all without interaction with the user (points i and ii above).

Interacting with the user (point iii) is made as simple as possible. The Investigator is able, to some extent, to take into account global goals specified by the user. The Assistant allows users to start Investigator using manually compiled datasets and specify an overall investigation goal by selecting an appropriate strategy. Results of the analysis could be exploited by the user, in particular user can save any produced collection, edit it and analyse with any processor. In the current implementation of the Demonstrator, the computational graphs produced by the users are technically separate from those made by the Investigator and do not support online interaction at the level of the graphs. The internal structures of the Investigator allow relatively smooth implementation of this functionality if later needed.

In collaboration with WP7 (demonstration, dissemination, outreach and exploitation), we have integrated the Personal Research Assistant into the Demonstrator. A significant amount of effort has been invested to coordination between the Assistant and the Demonstrator. In addition to its main functionality, the Demonstrator currently includes a special API which allows the Investigator to query definitions of user-defined datasets and compound articles made by users.

We also collaborated with historians working on the WP6 (digital humanities, applications and uses). They provided feedback on the Personal Research Assistant during regular tool testing sessions, which resulted in Deliverable 6.9 (see Section 2.5). Historians appreciated the general idea of the Personal

Research Assistant ‘as the tool bringing all research steps together and explaining them to the user in a meaningful way’. However, they mentioned that ‘the set of tools available is currently limited’ and ‘its functionality is not fully integrated into the Demonstrator’. External users had an opportunity to test the Personal Research Assistant during the NewsEye User Workshop organized online on April 13th 2021³. The Demonstrator and the Personal Research Assistant are freely available online⁴, as well as the location vizualization tool⁵.

³<https://www.newseye.eu/user-workshop/save-the-date/>

⁴<https://github.com/NewsEye/>

⁵<https://github.com/dhh21/SpaceWars>

References

- [1] Gary Marchionini and Ryen White. “Find what you need, understand what you find”. In: *International Journal of Human-Computer Interaction* 23.3 (2007), pp. 205–237.
- [2] Ryen W White and Resa A Roth. “Exploratory search: Beyond the query-response paradigm”. In: *Synthesis lectures on information concepts, retrieval, and services* 1.1 (2009), pp. 1–98.
- [3] Lindsay C. Page and Hunter Gehlbach. “How an Artificially Intelligent Virtual Assistant Helps Students Navigate the Road to College”. In: *AERA Open* 3.4 (2017), p. 2332858417749220.
- [4] Katharina Keller, Kim Valerie Carl, Hendrik Jöntgen, Benjamin M. Abdel-Karim, Max Mühlhäuser, and Oliver Hinz. ““K.I.T.T., Where Are You?": Why Smart Assistance Systems in Cars Enrich People's Lives". In: *Adjunct Proceedings of the 2019 ACM International Joint Conference on Pervasive and Ubiquitous Computing and Proceedings of the 2019 ACM International Symposium on Wearable Computers*. UbiComp/ISWC '19 Adjunct. London, United Kingdom: ACM, 2019, pp. 1120–1132. ISBN: 978-1-4503-6869-8.
- [5] B. Sateli, E. Angius, and R. Witte. “The ReqWiki Approach for Collaborative Software Requirements Engineering with Integrated Text Analysis Support”. In: *2013 IEEE 37th Annual Computer Software and Applications Conference*. July 2013, pp. 405–414.
- [6] Karen Myers, Pauline Berry, Jim Blythe, Ken Conley, Melinda Gervasio, Deborah L. McGuinness, David Morley, Avi Pfeffer, Martha Pollack, and Milind Tambe. “An Intelligent Personal Assistant for Task and Time Management”. In: *AI Magazine* 28.2 (June 2007), p. 47.
- [7] Angel Ivanov and Daniela Orozova. “Virtual Intelligent Personal Assistant for Bat Researchers”. In: *Proceedings of the 19th International Conference on Computer Systems and Technologies*. CompSysTech'18. Ruse, Bulgaria: ACM, 2018, pp. 38–41. ISBN: 978-1-4503-6425-6.
- [8] Richard B Segal and Jeffrey O Kephart. “Swiftfile: An intelligent assistant for organizing e-mail”. In: *In AAAI 2000 Spring Symposium on Adaptive User Interfaces*. Stanford, CA, 2000.
- [9] João Santos, Joel J.P.C. Rodrigues, Bruno M.C. Silva, João Casal, Kashif Saleem, and Victor Denisov. “An IoT-based mobile gateway for intelligent personal assistants on mobile health environments”. In: *Journal of Network and Computer Applications* 71 (2016), pp. 194–204. ISSN: 1084-8045.
- [10] Jaspreet Singh, Wolfgang Nejdl, and Avishek Anand. “Expedition: a time-aware exploratory search system designed for scholars”. In: *Proceedings of the 39th International ACM SIGIR conference on Research and Development in Information Retrieval*. ACM. 2016, pp. 1105–1108.
- [11] Jani Marjanen, Jussi Kurunmäki, Lidia Pivovarova, and Elaine Zosa. “The expansion of isms, 1820-1917: Data-driven analysis of political language in digitized newspaper collections”. In: (2020).
- [12] Stephanie Brandl and David Lassner. “Times Are Changing: Investigating the Pace of Language Change in Diachronic Word Embeddings”. In: *Proceedings of the 1st International Workshop on Computational Approaches to Historical Language Change*. Florence, Italy: Association for Computational Linguistics, Aug. 2019, pp. 146–150.
- [13] Tanja Säily, Eetu Mäkelä, and Mika Hämäläinen. “Explorations into the social contexts of neologism use in early English correspondence”. English. In: *Pragmatics & Cognition* 25.1 (2018). The Dynamics of Lexical Innovation: Data, methods, models. Special issue of Pragmatics & Cognition 25:1 (2018). Edited by Daphné Kerremans, Jelena Prokić, Quirin Würschinger and Hans-Jörg Schmid, pp. 30–49. ISSN: 0929-0907.

- [14] Nicolas Gutehrlé, Oleg Harlamov, Farimah Karimi, Haoyu Wei, Axel Jean-Caurant, Lidia Pivovarova, et al. “SpaceWars: A Web Interface for Exploring the Spatio-temporal Dimensions of WWI Newspaper Reporting”. In: *Proceedings of the 6th International Workshop on Computational History (HistoInformatics 2021)*. CEUR-WS. org. 2021.
- [15] J. O. Kephart and D. M. Chess. “The vision of autonomic computing”. In: *Computer* 36.1 (Jan. 2003), pp. 41–50. ISSN: 1558-0814. DOI: [10.1109/MC.2003.1160055](https://doi.org/10.1109/MC.2003.1160055).
- [16] Llorenç Escoter, Lidia Pivovarova, Mian Du, Anisia Katinskaia, and Roman Yangarber. “Grouping business news stories based on salience of named entities”. In: *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 1, Long Papers*. 2017, pp. 1096–1106.

A. API specification

The REST API has four main entry points described below: *investigator*, for interactions with the Investigator, *analysis*, for running text analysis tools manually, *report*, to wrap the Reporter calls, and *explainer*, to call the Explainer.

investigator

GET /api/investigator/

Retrieve all investigator runs started by the user

POST /api/investigator/

Start a new investigator run, and return its basic information to the user. Takes as an input a collection definition.

GET /api/investigator/result{?run=run_uuid}

Retrieve results for an investigator run.

analysis

GET /api/analysis/processors/

Retrieve information on the available processors

POST /api/analysis/

Start a new analysis task, and return its basic information to the user

GET /api/analysis/{task_uuid}

Retrieve results for an analysis task

report

GET /api/report/formats

List the text formatting options supported by the Reporter component

GET /api/report/languages

List the languages supported by the Reporter component

GET /api/report/report{?task=task_uuid|?run=run_uuid}{?nolink}

Retrieve a report generated from the task results; *nolink* parameter is used to disable links from report text to other objects in the API.

explainer

GET /api/report/formats

List the text formatting options supported by the Explainer component

GET /api/report/languages

List the languages supported by the Explainer component

GET /api/report/report{?task=task_uuid|?run=run_uuid}

Retrieve an explanation generated from the task results

B. SpaceWars: A Web Interface for Exploring the Spatio-temporal Dimensions of WWI Newspaper Reporting

Published as Nicolas Gutehrle et al. "SpaceWars: A Web Interface for Exploring the Spatio-temporal Dimensions of WWI Newspaper Reporting". In: *Proceedings of the 6th International Workshop on Computational History (HistInformatics 2021)*. CEUR-WS. org. 2021

SpaceWars: A Web Interface for Exploring the Spatio-temporal Dimensions of WWI Newspaper Reporting

Nicolas Gutehrle¹, Oleg Harlamov², Farimah Karimi³, Haoyu Wei⁴,
Axel Jean-Caurant⁵ and Lidia Pivovarov⁴

¹University of Franche-Comté, France

²Friedrich-Alexander University Erlangen-Nürnberg, Germany

³University of Cologne, Germany

⁴University of Helsinki, Finland

⁵University of La Rochelle, France

Abstract

In this paper, we describe an interactive map that places automatically extracted location names on a map and allows historians to study spatial imaginaries across time and newspapers. Our contribution is two-fold: first, we present a working instrument for historical studies that is freely available on the web; second, we describe a data analysis pipeline, which can also be applied to other data and material. Challenges we address in this paper range from handling textual noise introduced by Optical Character Recognition (OCR) and Named Entity Recognition (NER) applied to historical documents, georeferencing textual place mentions, and issues connected with web design and the ultimate user experience.

Keywords

Named Entities, visualization, georeferencing, newspapers, WWI

1. Introduction

The number and size of digitized text collections is continually growing leading to a rise of interest in macroanalysis or distant reading methods[1] We offer a tool for historians that visualizes the locations mentioned in newspapers on a map and provides various functions to analyze spatio-temporal information, thus simplifying studies of spatial imaginaries.


Spatial imaginaries are "stories and ways of talking about places and spaces that transcend language as embodied performances by people in the material world" [2]. Spatial imaginaries are shared by large groups of people, or a society as a whole [3]. Media play a vital role in reflecting public discourses, including spatial imaginaries. This could be seen in ways they are referring to different locations and, even more importantly, in the amount of attention they pay to various places. Huge events, such as wars, transform spatial imaginaries in many ways. This

HistInformatics 2021 – 6th International Workshop on Computational History, September 30, 2021, online

✉ nicolas.gutehrle@univ-fcomte.fr (N. Gutehrle); oleg.harlamov@fau.de (O. Harlamov);
karimi.farimah@gmail.com (F. Karimi); hayou.wei@helsinki.fi (H. Wei); axel.jean-caurant@univ-lr.fr
(A. Jean-Caurant); lidia.pivovarov@helsinki.fi (L. Pivovarov)



© 2021 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

 CEUR Workshop Proceedings (CEUR-WS.org)

transformation is reflected in and accelerated by the media and thus can be observed in large collections of historical newspapers.

Even though automatic analysis of massive text collections can help reveal hidden patterns in the data and lead to new research questions, it is not sufficient to study the construction of spatial imaginaries. As stated by [4], it is necessary to go back to the source document and study these location mentions in their context combining distant and close readings of the documents. Thus, in addition to maps that provide a summary view of the data, our tool provides concordances for locations and links locations to the articles mentioning them, allowing historians to alternate massive automatic analysis with close reading.

The newspapers annotated with automatically extracted locations are provided by the NewsEye project [5], aimed at providing analytical tools for historical analysis of the past media. We limit our tool with the period of the First World War. NewsEye collection contains newspapers from Austria, France and Finland, which belonged to different parties during WWI and got their news from different channels.

In most of the texts the locations have been extracted and tagged. However, macroanalytical tools within NewsEye are not centered around locations. Visualizing spatial entities from a text collection in a geographical information system is a type of digital media transformation that puts the focus on the geospatial aspects of the data. Our web app thus offers a feature that was missing from the NewsEye platform and that enables users to explore the perception and influence of space and place in relation to other dimensions of the Great War¹.

We describe all steps performed to develop a web tool. Our code is publicly available² thus our approach can be reused for other historical use-cases. The remainder of the paper is organized as follows: Section 2 introduces some theoretical studies in the field of Geo-Humanities and related mapping projects. In Section 3 the underlying dataset is described, Section 4 outlines the data preprocessing and in Section 5 the web interface is presented.

2. Related Work

There have been extensive theoretical studies in the field of spatial digital humanities and on the methods of geographical text analysis and digital mapping [6, 1, 7]. [1] focuses on the conceptual modelling and transformation of textual data to the medium of maps, while [7] offer a collection of essays on the influence of the use of geospatial analysis in the discipline of literary studies and showcases a number of literary mapping projects. [8] propose a methodology to extract "spatial-temporal profiles" which list normalised temporal and spatial expressions that co-occur in documents. Such profiles can then be used to visualise on a map the succession of events as depicted in the document in chronological order. The ELEVATE-live web interface (<https://elevate.greyc.fr/>) by [9] shows the "virality" of news article across countries through map visualization. This interface relies on previous works [10], which demonstrates that the inherent semantics of documents can be explored thanks to information at the entity-level.

Next to projects revolving around literary and fictional texts (e.g., *A Literary Atlas of Europe* (<http://www.literaturatlas.eu/en/index.html>), *A Map of Paradise Lost* ([¹The web interface is available at <http://spacewars.newseye.eu/>](https://olvidalo.github.io/paradise-</p></div><div data-bbox=)

²<https://github.com/dhh21/SpaceWars>

lost/), there are also various examples of projects based on non-literary historical data. The *al-Turayyā* Project(<https://althurayya.github.io/>) is a gazetteer and a geospatial model of the early Islamic world visualizing over 2,000 toponyms and route sections from Georgette Cornu's *Atlas du monde arabo-islamique à l'époque classique: IXe-Xe siècles* (Leiden: Brill, 1983) on an interactive map with additional path finding and search features. A similar but much larger project is *ORBIS: The Stanford Geospatial Network Model of the Roman World* (<https://orbis.stanford.edu/>), which "reconstructs the time cost and financial expense associated with a wide range of different types of travel in antiquity" [11].

Running Reality(<https://www.runningreality.org/>) is an expansive research project that aims to model the evolution of human civilization. The spatio-temporal system is based on a complex world history model and can render any day of any year as a map that allows users to interact with and gain information about the displayed spatial entities and objects. *Trading Consequences*(<http://trcons.edina.ac.uk/vis/tradConVis/>) uses text-mining software to explore historical documents related to international commodity trading in the British Empire during the 19th century, and its impact on the economy and environment. The web map utilizes a combination of text-based and graphic visualization techniques like maps, diagrams and word clouds to present the information extracted from the documents.

[4] applies distant reading methods to study how the *Houston Daily Post* between 1984 and 1901 shaped what he terms an *imagined geography* by privileging places over others. Blevins approximates this privilege ranking of place mentions via the corresponding occurrence frequencies in this newspaper collection(<http://spatialhistory.stanford.edu/viewoftheworld>). These maps revealed that local places such as Houston, Dallas or Austin are much more frequent in the *Houston Daily Post* than more thriving cities at the time such as New-York or Chicago. This suggests that, at a time where the United States underwent an important process of nationalization and standardization, the newspaper actively emphasized its regional space.

Similarly to [4], we developed a web interface to explore how newspapers published during WWI created spatial imaginaries. Through this app, the user can explore the whole dataset or focus on specific newspapers, languages or time periods. The interface consists of two parts which we describe below: the map module and the concordancer module. The main difference between our project and others is that our application directly connects occurrences of place names to their corresponding geospatial entities visualized on the map.

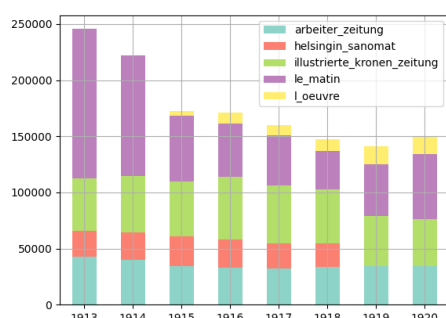
3. Dataset

3.1. NewsEye collection

The underlying data³ consist of automatically recognized and NER-tagged historical newspapers. Whenever possible, extracted entities have been automatically linked to a Wikidata resource. The details of the method can be found in [12, 13], and other project publications.

We use a subset published between 1913 and 1920. It covers 5 different newspapers from 3 different European languages: *Arbeiter Zeitung*, *Illustrierte Kronenzeitung* (German), *Le Matin*,

³The data have been provided by courtesy of the NewsEye project: <https://www.newseye.eu/>; for details, see [5]



(a) Unique toponym counts per newspaper-year



(b) Excerpt from the *Neue Freie Presse*, 11.08.1914

L'Œuvre (French), *Helsingin Sanomat* (Finnish). The number of unique location names for each year is shown in Figure 1a. Note that this statistics reflects peculiarities of the dataset rather than historical processes directly: e.g. *L'Œuvre* for 1913 and 1914, and *Helsingin Sanomat* for 1919 and 1920 are missing even though the newspapers were published in these periods. We still need to explain the larger number of unique names in 1913 and 1914; one possible explanation might be a worse OCR quality that leads to producing more name variants.

The quality of the OCR and NER varies across newspaper issues and languages, which creates major challenges for assessing the accuracy of the character strings recognized and classified as place names. For example, in Figure 1b we show an excerpt from an Austrian newspaper article announcing a war between Britain and Germany. However, the NER system recognized only one location in this text, namely Vienna, which is the publication location rather than a location of the event. For NewsEye collection, the NER F-measure varies from 57.13% (French) to 36.39% (Finnish), the Named Entity Linking (NEL) F-measure varies from 48.4% to 30.0%. However, the numerical results were obtained using small subsets of the data, since there are no manually annotated historical datasets for these tasks. Such problems are common for historical datasets [14, 15] and explains why automatic processing is usually accompanied by close reading.

3.2. External reference

To compare location coverage in media with places where events took place in reality, we completed our dataset with data relating to battles during the WWI.

We are unaware of any professional database providing access to battle locations in a machine-readable form and thus rely on Wikidata to obtain this information. Wikidata contains well-structured battle informations and saves them as a semantic frameworks. We extracted battles from Wikidata using SPARQL with the *WWI* keyword as a query. For each battle, we extracted its name and coordinates, start and end dates, the war fronts it belongs to (e.g. Western), the country where it took place, and its duration in days. The ontology of war fronts on Wikidata is illustrated in Figure 2. The WWI ontology has a structured hierarchy, most of the battles are linked to a specific front, some are linked to WWI directly, and some of the battles are linked to

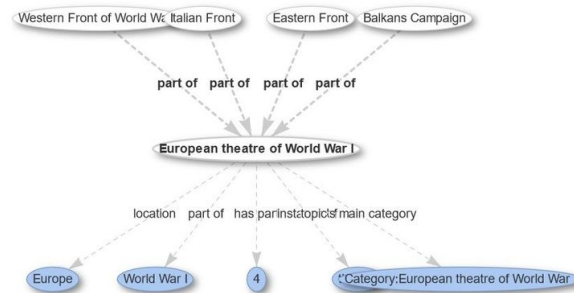


Figure 2: WWI war theatre ontology.

a subpart of the WWI ontology (e.g. Finnish civil war). We make sure we had extracted all the battles related to WWI, direct or indirect. One of our findings is that WWI data on Wikidata are unbalanced. The European war theater is much more developed than other parts of the world, and even within the European war theater, the Western front is more developed than others (e.g. Balkans Campaign or Italian front). For the less developed parts, the data can be incomplete, have wrong or even missing coordinates. In some cases we fixed them and cleaned manually for our app, though we did not have resources to add complete information from other sources. Nevertheless, adding battle data on the app can be useful, since it helps users to place anchor location references into the map.

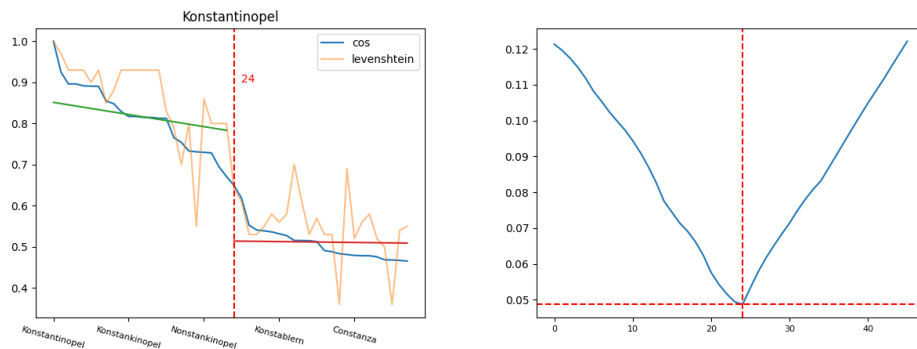
4. Data preprocessing

4.1. Entity mention normalization

In order to address the OCR-induced orthographic variation, we adopted a partial-matching-based approach [14, 16], i.e. finding candidate spellings via feature-vector similarity comparisons. According to [17], character-ngram representations provide significant improvements over word vectors for French (14%), German (28%) and Finnish (66%) monolingual retrieval in terms of *mean average precision*. Moreover, [17] establishes a high correlation (0.935) of this accuracy boost with the mean word length per language, thereby indicating its pertinence for morphologically more complex languages, e.g. Finnish and German.

Concretely, character-level n-gram counts are extracted for all unique entity tokens of a given newspaper-year domain, i.e. each entity mention is represented by an n-gram vector. In a next step, the top 100 most frequent mentions are used as query terms over the total set of unique mentions of the underlying newspaper-year domain. Given the resulting sparse vector representations, cosine distance has been employed due to its robustness w.r.t. vector dimensionality and successful application to syntactic matching problems [18].

Ultimately, a cut-off threshold needs to be determined for the most salient spelling variants in the cosine ranking w.r.t the query term. To achieve this, we apply the extended *L-method*, a fully automatic separation algorithm for large evaluation graphs [19]. The rationale behind this method lies in the assumption that evaluation graphs can be approximated by means of two



(a) Extended L method applied on the query term *Konstantinopel* (b) Combined RMSE loss function optimal cutoff for query term *Konstantinopel*.

Figure 3: Most salient nearest neighbour retrieval via extended *L-method*

regression lines L_c and R_c separated by a cutoff point c that minimizes the linearly interpolated objective functions for both sides of the cutoff point:

$$L(c) = \frac{c-1}{n-1}RMSE(L_c) + \frac{n-c}{n-1}RMSE(R_c) \quad (1)$$

where n is the total length of the set of unique place mentions in the newspaper-year domain and RMSE stands for Root-Mean-Square Error.

For example, Figure 3a shows name mentions ranked according to their bag-of-ngram cosine similarity to the query term *Konstantinopel*. The method successfully captures OCR-conditioned near-duplicates e.g. *Konstankinopel*, *Nonstankinopel* and distinguishes these from dissimilar terms, e.g. *Konstablern*, *Constanza*. The objective function is depicted in Figure 3b where the optimal cut-off point is visualized by the vertical dashed red lines in both plots. The optimal regression lines on the left and right of the cut-off point are highlighted in green and red respectively in Figure 3a. In our case of particularly large evaluation graphs the algorithm is applied iteratively with each computed cutoff point serving as the basis for the respectively consecutive iteration until convergence [19].

4.2. Georeference extraction

Subsequently, the normalized place mentions are mapped onto a corresponding cartographic representation. Following Yao's theoretical concept of *geocoding* as an act of *discrete georeferencing* [20], we fall back on *OpenStreetMap* data served by the *Nominatim* open-source georeferencing service⁴. Dedicated historical map services e.g. *OpenHistoricalMaps*⁵ or *RunningReality* could not be used because of a lack of Application Programming Interface (API).

⁴<https://nominatim.org/release-docs/develop/develop/Ranking/>

⁵*OpenHistoricalMap* <https://openhistoricalmap.org>

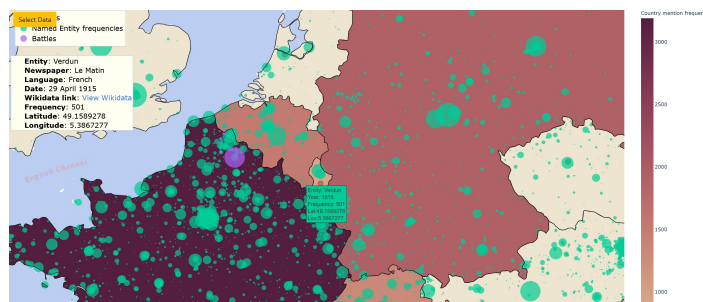


Figure 4: Visualizing every entity in *Le Matin* (1913-1915) through the map module

While contemporary maps do not account for historical geographical entities, such as *Austria-Hungary* or *Ottoman Empire*, we leverage the statistical properties of the dataset and the fact that major war events tend to appear in clusters around less prominent, peripheral locations in the vicinity of country borders, e.g. Western Front. We therefore argue that the eventual density and number of cartographic events serve as a proxy confidence for the geotagger output. Thus, the development and provision of such utilities remains a consideration for future work.

5. Web interface

5.1. Map navigation module

We created a base map that shows the borders and the capitals of countries between 1913 and 1920. This map is based on [21] which combines multiple sources such as [22] or the *Territorial Change Dataset* by [23] in order to represent country borders from 1886 to 2019. On top of the base map the tool shows frequencies of the locations mentioned in the data subset the user has selected. The tool can show the full dataset or filter it by language, newspaper title or year.

The map is exemplified in Figure 4. The more frequent a location is in the selected dataset the bigger it appears on the map. Each country on the map is associated with a colour: the darker the colour, the more frequent that country. The map also shows contextual information such as capital cities and battles that occurred in the selected time period. The longer a battle lasted and the bigger it will appear on the map. Hovering over or clicking on an entity will show the metadata related to it such as its frequency, the newspaper it appears in or the link to the Wikidata resource associated with that entity mention if that link has been found⁶.

It can be seen in Figure 4 that most named entities mentioned in *Le Matin* between 1913 and 1915 are located in France, as well as in Germany, Switzerland and Belgium. Those countries are also more frequent than other surrounding countries. This would suggest that *Le Matin* had a Eurocentric view of the war and focused on the Western front, even though the fighting spread also to Africa and Asia. This is an example of biases that are relevant for historical research and could be easily found with our interface.

⁶More details on the interface can be seen in our tutorial video:<https://youtu.be/iIpEvM9IFaM>

notre troisième armée et la	place de Verdun	. Violemment contre-attaqués, ils ne
ves de la	place de Verdun	. En avant de cet
	Verdun	;
	Verdun	était intact et l'armée française

Figure 5: Concordancer table view

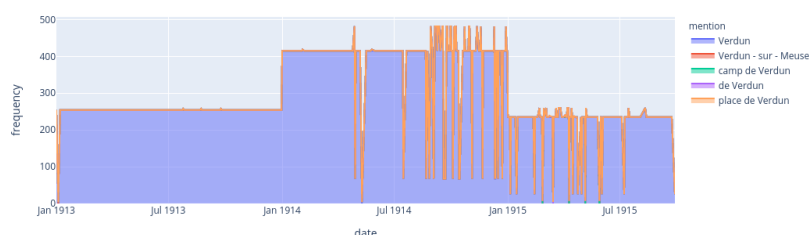


Figure 6: Concordance time series view

5.2. Concordancer module

To enable close reading functionality, we implemented a concordancer that shows every mention of a location surrounded by its left and right context. By default, these contexts are limited to 5 words before and after the mention. Each mention in the table is associated with a link that redirects the user to the NewsEye platform. There the user can extend the research by reading the article and the newspaper where an entity mention appears. The concordancer is linked to a plot that shows the distribution of entity mentions across time, by language or by newspaper.

Figure 5 shows every occurrences of “Verdun” in *Le Matin*(1913-1915) while Figure 6 show its distribution across that period. Because of its proximity with the German border, Verdun was a key position in the conflict. Thus, it is of no surprise that Verdun is nearly exclusively mentioned in war reports or articles related to the conflict. Interestingly, other location mentions related to Verdun reinforce the relation of the city with war. For instance, Verdun is sometimes mentioned as “place de Verdun”. “place de” usually indicates a city square, however in this case, “place de Verdun” always refers to the city of *Verdun* itself. There are also mentions of the military camp in Verdun (“camp de Verdun”), but only in articles published in 1913 referring to the Franco-Prussian war. Both these mentions insists on the importance of the city as a defensive position, even before the beginning of WWI. The only exception to that is the mention of “Verdun-sur-Meuse”, which was the official name of the city between 1801 until 1970 and which is only mentioned to indicate the birthplace of a person.

6. Conclusion

We presented a publicly available web application aimed at supporting analysis of spatial imaginaries that are reflected in historical newspapers published during WWI. The data, provided by the NewsEye project, include automatically extracted location names, which we cleaned up

and mapped to geographical coordinates. The interface summarizes large amounts of data and enables the comparison of constructed imaginary spaces emanating from place name collocations with the physical spatial dimensions of WWI. The combination of map, concordancer and original links to articles allows an easy switching between distant and close reading.

Our experiments also reveal the lack of machine-readable representations of historical knowledge, such as historically-aware gazetteers, historical event locations (e.g. battle places) and other information. Thus, even though WWI is exhaustively studied in historiography, resorting to community-driven resources, e.g. *Wikidata*, as part of the *Wikimedia Commons*, rather than dedicated scholarly ones was largely driven by technical convenience.

We believe this problem is common for many use cases and should be addressed in the future by creating shared historical knowledge bases with open access points in order to facilitate the development of analysis tools for historical research. In addition, many resources that allow users to browse historical information online do not provide utilities for automatic reuse of the data. The development and provision of such utilities remains a consideration for future work.

Acknowledgments

This work has been supported by the European Union Horizon 2020 research and innovation programme under grant 770299 (NewsEye), and by the Région Bourgogne Franche-Comté, France, as part of the EMONTAL project.

References

- [1] Ø. Eide, *Media Boundaries and Conceptual Modelling: Between Texts and Maps*, Palgrave Macmillan UK, 2015. URL: <https://www.springer.com/de/book/9781137544575>. doi:10.1057/9781137544582.
- [2] J. Watkins, Spatial imaginaries research in geography: Synergies, tensions, and new directions, *Geography Compass* 9 (2015) 508–522.
- [3] S. Davoudi, J. Crawford, R. Raynor, B. Reid, O. Sykes, D. Shaw, Spatial imaginaries: tyrannies or transformations?, *Town Planning Review* (2018).
- [4] C. Blevins, Space, nation, and the triumph of region: A view of the world from houston, *Journal of American History* 101 (2014) 122–147. doi:10.1093/jahist/jau184.
- [5] A. Doucet, M. Gasteiner, M. Granroth-Wilding, M. Kaiser, M. Kaukonen, R. Labahn, J.-P. Moreux, G. Muehlberger, E. Pfanzelter, M.-E. Therenty, et al., Newseye: A digital investigator for historical newspapers, in: 15th Annual International Conference of the Alliance of Digital Humanities Organizations, DH 2020, 2020.
- [6] D. J. Bodenhamer, J. Corrigan, T. M. Harris (Eds.), *The Spatial Humanities: GIS and the Future of Humanities Scholarship*, Indiana University Press, Bloomington & Indianapolis, 2010.
- [7] D. Cooper, C. Donaldson, P. Murrieta-Flores (Eds.), *Literary mapping in the digital age, Digital research in the arts and humanities*, first published ed., Routledge, Taylor & Francis Group, London New York, 2016.

- [8] J. Strötgen, M. Gertz, P. Popov, Extraction and exploration of spatio-temporal information in documents, in: GIR, 2010.
- [9] Govind, C. Alec, M. Spaniol, Elevate-live: Assessment and visualization of online news virality via entity-level analytics, in: ICWE, 2018.
- [10] Govind, M. Spaniol, Elevate: A framework for entity-level event diffusion prediction into foreign language communities, Proceedings of the 2017 ACM on Web Science Conference (2017).
- [11] W. Scheidel, Orbis: The Stanford Geospatial Network Model of the Roman World, SSRN Electronic Journal (2015). URL: <http://www.ssrn.com/abstract=2609654>. doi:10.2139/ssrn.2609654.
- [12] E. Boroş, A. Hamdi, E. L. Pontes, L.-A. Cabrera-Diego, J. G. Moreno, N. Sidere, A. Doucet, Alleviating digitization errors in named entity recognition for historical documents, in: Proceedings of the 24th Conference on Computational Natural Language Learning, 2020, pp. 431–441.
- [13] E. Linhares Pontes, A. Hamdi, N. Sidere, A. Doucet, Impact of ocr quality on named entity linking, in: Digital Libraries at the Crossroads of Digital Information for the Future, Springer LNCS, 2019, pp. 102–115. URL: <https://doi.org/10.5281/zenodo.3529180>. doi:10.5281/zenodo.3529180.
- [14] S. Dumais, Improved String Matching Under Noisy Channel Conditions, in: Proceedings of CIKM 01, 2001, pp. 357–364. URL: <https://www.microsoft.com/en-us/research/publication/improved-string-matching-under-noisy-channel-conditions/>.
- [15] P. Kantor, E. Voorhees, The trec-5 confusion track: Comparing retrieval methods for scanned text, Information Retrieval 2 (2000) 165–176. doi:10.1023/A:1009902609570.
- [16] R. Jin, C. Zhai, A. Hauptmann, Information retrieval for ocr documents: A content-based probabilistic correction model, Proceedings of SPIE - The International Society for Optical Engineering 5010 (2003) 128–135. doi:10.1117/12.472838, document Recognition and Retrieval X ; Conference date: 22-01-2003 Through 24-01-2003.
- [17] P. Mcnamee, J. Mayfield, Character n -gram tokenization for european language text retrieval, Information Retrieval 7 (2004) 73–97. doi:10.1023/B:INRT.0000009441.78971.be.
- [18] R. S. Mishra, K. Mehta, N. Rasiwasia, Scalable approach for normalizing e-commerce text attributes (SANTA), CoRR abs/2106.09493 (2021). URL: <https://arxiv.org/abs/2106.09493>. arXiv:2106.09493.
- [19] S. Salvador, P. Chan, Determining the number of clusters/segments in hierarchical clustering/segmentation algorithms, in: 16th IEEE International Conference on Tools with Artificial Intelligence, 2004, pp. 576–584. doi:10.1109/ICTAI.2004.50.
- [20] X. Yao, Georeferencing, geocoding, in: R. Kitchin, N. Thrift (Eds.), International Encyclopedia of Human Geography, Elsevier, Oxford, 2009, pp. 458–465. URL: <https://www.sciencedirect.com/science/article/pii/B978008044910400448X>. doi:<https://doi.org/10.1016/B978-008044910-4.00448-X>.
- [21] G. Schvitz, S. Rüegger, L. Girardin, L.-E. Cederman, N. Weidmann, K. S. Gleditsch, Mapping The International System, 1886-2017: The CShapes 2.0 Dataset, Journal of Conflict Resolution (2021). URL: <https://journals.sagepub.com/doi/full/10.1177/00220027211013563>. doi:10.1177/00220027211013563.
- [22] K. S. Gleditsch, M. D. Ward, Interstate system membership: A revised list of the independent

states since 1816, *International Interactions* 25 (1999) 393–413.

- [23] J. Tir, P. Schafer, P. F. Diehl, G. Goertz, Territorial changes, 1816–1996: Procedures and data, *Conflict Management and Peace Science* 16 (1998) 89–97. URL: <https://doi.org/10.1177/073889429801600105>. doi:10.1177/073889429801600105. arXiv:<https://doi.org/10.1177/073889429801600105>.