Group 50 – Clifton Roozendal, Floris ten Lohuis, Jens van Holland

First Assignment  
Entity Linking

WEB DATA PROCESSING SYSTEMS

Contents

[Introduction 1](#_Toc88834730)

[Setup 1](#_Toc88834731)

[Fundamental Choices 1](#_Toc88834732)

[Program 1](#_Toc88834733)

[Main 2](#_Toc88834734)

[Pipeline 2](#_Toc88834735)

[Clean 2](#_Toc88834736)

[Extract 3](#_Toc88834737)

[Search 3](#_Toc88834738)

[Decision 4](#_Toc88834739)

[Next Steps / Future Releases 4](#_Toc88834740)

[Difficulties Encountered 4](#_Toc88834741)

# Introduction

In many real-world and theoretical use cases it is very useful to recognize entities mentioned in a given text and link these to a predefined Knowledge Graph. This not only helps to relate knowledge from a variety of source to each other but can also give a better contextual understanding of the topic and most important entities covered in the text. Accomplishing this type of recognition requires several (pre-)processing steps, such as cleaning and standardising the (semi-)unstructured data, removing information which adds little value and finally passing this processed data through a knowledge graph in order to recognize the most important entities in the data. The program which will be described in this document aims to perform exactly these tasks. The document will describe the setup of the various modular steps being performed and also aims to give a rationale behind why specific choices were made.

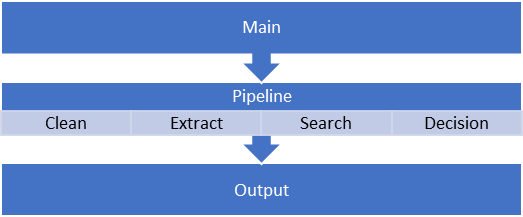
# Setup

## Fundamental Choices

There are several fundamental choices made in the program. First off, the entire program has been built in a docker container. This helps to localise the entire setup and allows for a coherent way to share the program and also work across different operating systems. The docker container being used has been provided for the assignment and already includes some pre-installed software, namely Trident and ElasticSearch. This brings us immediately to the next fundamental choice; the Wikidata Knowledge Graph which has been setup both in Trident and ElasticSearch on the docker container will be used to perform the entity linking. These repositories have all been set up locally in the docker container in order to optimise speed, efficiency and stability.

## Program

Having an understanding of the environment, we can now move to describing the program being run within it in order to perform the actual entity linking. The program has been written in Python and consists of several modular classes which each perform a given processing step. These modular classes are: Clean, Extract, Search and Decision and will be described individually in their own respective sections below. The classes are linked together in a separate Pipeline class which ensures that each class is called successively and validates that the correct inputs and outputs are passed to each of them. Finally, the entire pipeline is executed through a master file, Main. This master file is the entry point for the program. It gives the user the option to define various different arguments which are used throughout the entire code to parameterise processes or choose which ones should be used for the run.



### Main

The Main file is the entry point for the program. As mentioned above, it exposes several parameters used throughout the program to the end user. In order to keep the program user friendly, default values which are generically deemed to be fitting have been defined for each of them. The arguments are:

|  |  |  |
| --- | --- | --- |
| Argument | Default Value | Description |
| data\_dir | /app/assignment/data/sample.warc.gz | Directory of the xxxx.warc.gz file which will be used to for the entity linking. |
| clean\_text | 1 | The cleaning procedure which is used:\n\t1: html2text (default)\n\t2: BeautifulSoup:html.parser |
| extract\_model | en\_core\_web\_sm | The model which is used to extract:\n\ten\_core\_web\_sm (default)\n\ten\_core\_web\_lg |
| query\_size\_ES | 15 | The max number of hits a query can return |
| search\_ES | normal | Please do not alter this - the 'fast' implementation is very buggy |
| batch\_size\_NER | 8 | The NER model parses n samples in parallel.\nWe found 8 to be the best value (on 8 threads, intel i7, 16gb RAM) |
| n\_threads | 8 | The number of threads to use.\nPlease be carefull - do not set to -1, this will go wrong(!) |
| sim\_cutoff\_NER | 0.35 | To reduce the number of queries = entities, we compute similarity.cross-referencing scores (no time to implement that in parallel) and use a threshold (last one is kept) |

In addition to handling the arguments, the main file also sets up and executes the pipeline.

### Pipeline

The pipeline class can be seen as the backbone of the program as this is where the bulk of the functionality comes together and gets linked to each other. In addition to calling the 4 underlying sub-processes and validating that the output of a given sub-process is compatible with the input of the subsequent one, the pipeline also contains some other vital steps in the process.

The first step in the pipeline parses the WARC file. It then passes each parsed record to the sub-processes in the pipeline. Once all of the records have gone through the pipeline, it writes the results to a text file in the predefined format which is needed for scoring the results.

### Clean

The Clean class is the first processing step in the pipeline. The ultimate goal of the Clean function is to parse the WARC file, which is in html format, into a more useable string. There are two options for parsing the html text, either using the html2text or BeautifulSoup (bs4) python packages. The option can be defined by the end user using the clean\_text argument when calling the main function. Note that both packages work well for the given task without much parameterisation.

In order to optimise runtime, the process has been implemented to run in parallel (using the multiprocessing package) so that it can clean several records at once.

## Extract

The Extract class is the second step in the pipeline. It aims to extract important entities from the provided (cleaned) text. It does this in several steps.

The first step is to use natural language processing to extract the most important entities from the given text. To do this, the spacy package was used and loaded with either the “en\_core\_web\_sm” (English - small) or “en\_core\_web\_lg” (English - large) models. The model being used can be defined by the user with the extract\_model argument (the en\_core\_web\_sm was chosen as a default because it is much more efficient and performs almost as well as the large one). The spacy package is a sophisticated package which contains multiple pipelines to, for example, perform lemmatization, tok2vec and named entity recognition. However, for computational benefit and because it performed well for the given task, only the Named Entity Recognition (NER) pipeline of the spacy model was used.

Once the entities were extracted, these were further refined by, in succession, removing duplicates, removing certain characters within an entity, removing blacklisted entities and finally removing entities which were identified as being similar to other entities. Certain characters, such as “\”, “(” and multiple spaces were removed using regular expressions (RegEx). These were removed in an effort to further clean the entity strings. Blacklisted entities were defined during the building and testing phase of the model and were chosen because they generally seemed redundant for the task. Similarity scores were computed by iteratively comparing a given entity with all other entities. These scores were computed using the difflib package. If entities were computed to have a similarity score above a certain threshold (as defined by the sim\_cutoff\_NER argument, default 0.35), only one of them (the first in the list) was kept. This was again done to optimise computation time whilst, due to their computed similarity, not losing much information.

After all of the above steps were completed, a refined set of entities for each WARC record were found which could then be sent on to the Search class to link to other entities.

### Search

The aim of the Search class is to link the extracted entities to other entities in a Knowledge Graph. The Knowledge Graph being used in this program is Wikidata, which was loaded into a local ElasticSearch server to be efficiently queried. The ElasticSearch was queried for each WARC record in order to return a given number of records which were calculated to have the highest relevance scores. The number of records being returned was again a user defined parameter (query\_size\_ES) with a default value of 15. Note that this default cut-off was chosen as it yielded a manageable number of results and because the relevancy scores generally tapered off at around this number.

The program has two separate implementations of the ElasticSearch querying, one which is “normal” in the sense that it performs one query after the next and the other which is asynchronous. The main goal of the asynchronous path is to speed up the entire process, as the querying itself was by far the largest bottleneck in the pipeline. Nevertheless, the asynchronous querying did not perform as expected and often led to timeout issues. Therefore, whilst both options are still available (and can be chosen using the argument search\_ES), it is strongly advised to run using the normal option.

### Decision

The Decision class is the last part of the pipeline before the final results are returned. The main goal of this class is to refine the entities found in the Search module by removing those which are deemed to be incorrect. This decision is performed using the Trident package and is done in several steps.

First, for each WARC file we look at the number of wikidata hits found in the Search class. If only one or zero hits are found, it is evident that no further processing needs to be done; in the case of zero hits nothing can be removed meaning therefore no further refinement is necessary or possible and in the case of one hit, this hit will always be returned and therefore there is also no further refinement necessary. If more than one hit is found, these hits can be further refined. This is done by querying the hits on Trident in order to get the information from the hit. All of the information is then converted into a vector so that it can be scored. Finally, a threshold is set to only keep a given number of hits with the highest scores as being the most relevant for each record. This is the final output from the pipeline.

Note that the Decision unfortunately did not perform as desired and the choice was therefore made to disable this part of the pipeline. Nevertheless, it has been included as a lot of the logic in the class is valid and could be improved upon to fix the process in a later step. Concretely speaking, the disable in this case means that the input given to this class is immediately returned again as the output.

# Next Steps / Future Releases

The current version of the program performs quite efficiently, with scalability in mind, and is able to process the given input all the way through to returning the refined Wikilink hits. It has also been set up in a modular way so that specific parts of the pipeline can be further tuned without affected earlier or later parts of the calculation. Some of the things which could be further fine tuned are:

* Cleaning
  + Try other (custom) html parsers
* Extract
  + Extend the current RegEx rules for further cleaning
  + Use more parts of the Spacy pipeline
  + Try different extraction models
    - Both different models within Spacy as well as completely different NLP processors
  + Run similarity matching in parallel to improve speed
  + Finetune similarity cut-off
* Search
  + Fix asynchronous searches to optimise speed
  + Improve ElasticSearch query speed
  + Finetune ElasticSearch queries (for example look at whether combining multiple words in {bool: {should : [...]}} queries improves results)
  + Finetune number of hits being returned
* Decision
  + Get the class working
  + Tune the number of records which should be kept (or base it on a different criteria)

# Difficulties Encountered

During the building of the program we faced several difficulties which will be outlined in this section.

First is the hardware needed in order to run the entire program locally. A (Windows) machine with 8GB of RAM could not sufficiently handle a Docker container with ElasticSearch running and processing the amount of records in question. After some time, the docker container would consume all of the memory and cause the host computer to completely freeze. Furthermore, even using a computer with better specs (16GB of RAM) would eventually run into issues as the memory used by Vmmem increased over time. The presumption is that there is a memory leak somewhere. After much trial and error, restarting the docker container regularly was the only option which worked.

Secondly, we spent quite some time getting the docker container and more specifically the ElasticSearch server to work on everyones machines. This was largely due to the amount of technicality (such as the amount of allocated memory) needed to set up the environment. Nevertheless, we got this running on everyone’s machines and, after working with it for longer, did get more of a hang of how everything worked. This “issue” was therefore resolved as the project elapsed.

Third was the performance of the ElasticSearch queries; we frequently ran into timeout errors. The exact cause of this issue could also not really be pinpointed as they were intermittent and not always encountered. The presumption we had was that the filesystem cache was full and it would then starts swapping to the hard disk. As the program ran, we did notice that the query speed would generally improve and we were therefore able to continue.