Group 50 – Clifton Roozendal, Floris ten Lohuis

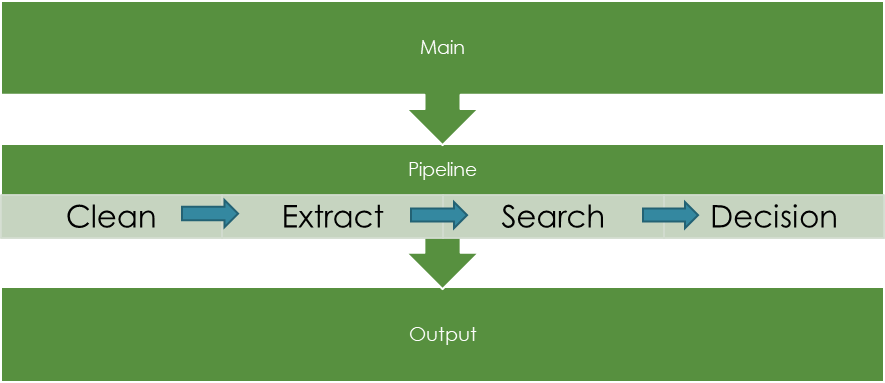
Second Assignment  
Entity Linking

WEB DATA PROCESSING SYSTEMS

# Introduction

In the first assignment where we built an entity linking program for HTML pages we had highlighted several possible areas of improvement (see appendix). For assignment 2 we therefore decided to focus on these areas in order to improve its robustnest, speed and performance. We had initially aimed to focus on the speed of the program but, considering the limited improvements we were able to realise here we decided to increase the scope and also focus on the other areas. This was also driven by the fact that better pre-processing could not only lead to better results but also improve the efficiency of the program by reducing the amount of unnecessary steps being performed.

The entity linking program was already set up as a pipeline with 4 main processing steps. These steps were: Clean, Extract, Search, Decision. In the below sections we will address the improvements made in each of these steps.



# Clean

In the cleaning part of the pipeline, which had the main goal of transforming the raw HTML pages from the WARC file into more understandable and processable text, we addressed several different improvements.

Firstly, we tested a variety of different html to text parsers to see if any of these would outperform the others for the given task. We ended up finding that BeautifulSoup provided the most flexibility and gave the best results. The main driver behind this was the fact that we could easily remove certain HTML elements such as script and style.

Secondly, we looked at how we could further clean the text. This was largely a recursive procress by looking at the outcomes of the cleaning as well as the later extract, search and decision steps to see where we still had unusable or unfound entities. We ended up further cleaning the text by:

* Remove meta data such as the HTTP header at the top of the page
  + In the initial implementation we found many entities which were simply found in this meta data and therefore not actually relevant)
* Replacing certain characters such as brackets with spaces
  + Quite a few entities were concatenated with each other or unusable as they contained unwanted seperators without a space
* Replace headlines into individual lines
  + We noticed that our NER model would perform better when various parts of the HTML page were more clearly split

# Extract

In the extract part of the pipeline the goal is to extract useful entities from the given text. Here we also focussed on a couple of areas.

Firstly, we tried various different packages such as Stanza, NLTK and spaCy. In the end we found the spaCy, which we had also used in assignment 1, was the most robust and best performing model and therefore decided to continue with this.

Nevertheless, we also found that a lot of the entities were not useful for the task at hand. We therefore further refined the processing by:

* Blacklisting certain named entity types
* Blacklisting certain words (either a full match or if it contained certain strings) which were not useful in the later steps
  + For example, entities containing http, png, jpg would often be extracted but almost never yielded any search results
* Extend RegEx to further clean entities
  + For example, we saw that many entities still contained special characters which would cause the search to be less effective
* Improve text similarity comparison
  + For every html page, extracted entities would be compared with each other to see if words were similar. We finetuned the similarity cutoff to be much more strict (cutoff from 0.35 to 0.75) as we realised that some entities were unrightfully being considered similar

In addition to improving the results we also rewrote the entire code to make it more efficient such as removing redundant for loops and instead performing all steps in one loop. We also made the extraction multi-threaded but did not gain much improvement versus using the spaCy pipe functionality (as this was already optimised for multi-threading).

# Search

The aim of the search function is to try and find the extracted entites in WikiData. During the first assignment this was really the bottleneck in the program and we therefore wanted to focus on speed improvements. In the end, we had two big improvements here:

* Created a cache which stored queries already performed. By doing this we reduced the number of unnecessary calls to ElasticSearch. We also tried several other
* By analysing which entites did not yield any search results we further finetuned the pre-processing so that these would be removed or better handled

We also tried to improve search by trying different and more complex queries such as match and bool but none of these yielded better results than the simply query.

Lastly, we also implemented ElasticSearch Async. We had expected this to yield quite a significant improvement but unfortunately this was not the case. We expect this to be largely due to the fact that the hardware we were running the program on was already the largest bottleneck and that ElasticSearch was therefore not able to fully utilise the memory it needed

# Decision

In the first assignment we were unfortunately not able to implement a working decision module. We therefore simply passed on all search results which had a strong negative effect on the outcome. For assignment 2 we therefore focussed on building logic which could help identify the correct entity from the given search results. This was a process which contained of several steps:

By comparing the type of the entity found in NER with that of the type found in the WikiData we were able to remove a lot of search results and instead focussed on getting the best match for each entity. By doing this we were able to remove a lot of false positive results which had a positive affect on the F1 score on which we were optimising.

# Analysis Report

Lastly, it is good to mention how we addressed the improvements in each of these areas. In order to gain more insights into the individual steps of the pipeline we exported intermediates at each step. We then built an analysis report on top of these outcomes so we could drill down and, for example, see which entities were extracted from the text, which search results were then found for these entities and how these were disambiguated in decision. By having this insight we could recursively implement and test further improvements. We found that this gave much more insight than looking at the F1 score on our set alone because the procedure was otherwise largely a black box.

A screenshot of a computer

Description automatically generated with medium confidence

# Appendix

## Next Steps / Future Releases

The current version of the program performs quite efficiently, with scalability in mind, and can process the given input 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 affecting 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 entirely different NLP processors
  + Run similarity matching in parallel to improve the speed
  + Fine-tune similarity cut-off
* Search
  + Fix asynchronous searches to optimize 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 different criteria)