Zhaolin Fang, Yunshan Wang, Tong Wu, Leonardo Kuffo

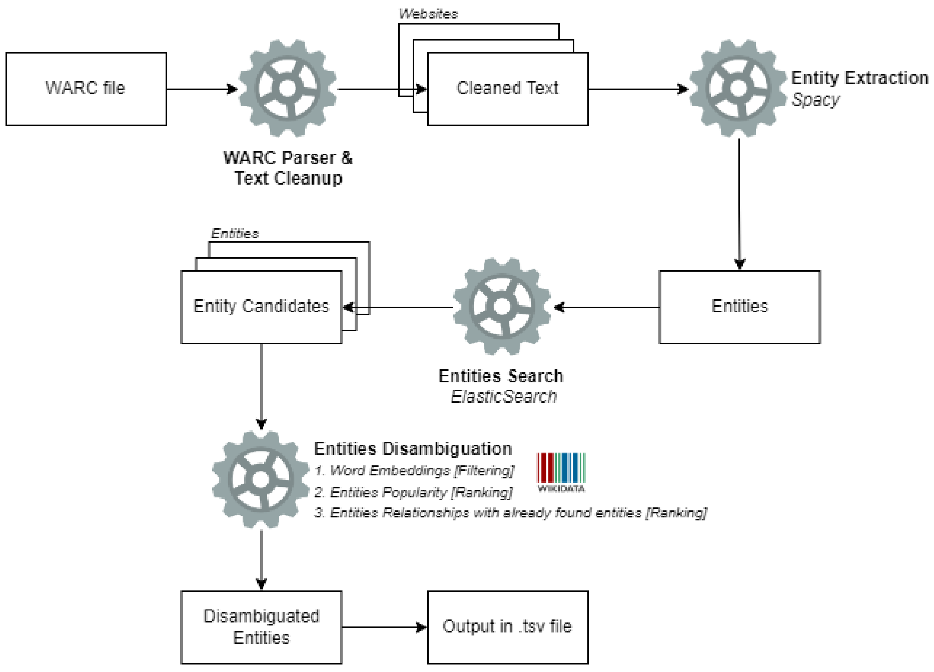
Assignment1: Webpage Entity Linker

1. Motivation

A large number of potential applications from bridging web data with knowledge bases have led to an incremental significance in the entity linking task. In this assignment we are going to implement an entity linker which is able to recognise entity mentions in a webpage and link them to the entities in a large Knowledge base (Wikidata).

To implement this entity linker, we developed a pipeline which contains several steps including WARC parsing, named entity recognition, entity candidates generation, candidates disambiguation and output the disambiguated entities with their website ID and their Wikidata ID.

2. Our Solution



As the diagram above shows, our pipeline firstly reads the WARC file to parse and clean the content, aiming to obtain cleaned text. Then the cleaned text will be passed to the entity extraction phase to recognize entities using NER tools. After this, these entities are used as input for ElasticSearch to query for entity candidates. Entity candidates are ranked and disambiguated before being written to the final output file. In the following sections, we will explain each stage of the pipeline in detail.

2.1. WARC Reading

In the first stage of our program, what we need to do is to design a WARC parser, which can obtain the html context from the WARC file. The WARC files contain the entire data source. Starting with a warcinfo record which illustrates the file, a range of records follow each warcinfo record. The most typical one is the request record that describes the way the content was requested. The response record describes what HTTP headers the server had sent. The metadata record contains other detailed information such as the detected languages, time to fetch the page and character sets.

To parse this WARC file we first split it into a sequence of lines in which each line represents a new record of WARC/1.0. Next, we use the warcio Python library as a helper to parse the file content. Inside this library, the ArchiveIterator class reads the WARC content into a single line (record) and then parses that line into WARCRecord objects. After this, we only proceed further if the record type of the WARCRecord is a response. From these records, we extract the HTML content stream. We ignored contents that were not HTML responses (e.g. XML or PDF content) by identifying the HTTP header Content-Type. In the end, we got lines of html as the result of the WARC parser. We also obtained the Record ID (WARC-TREC-ID) from the Record headers for further use.

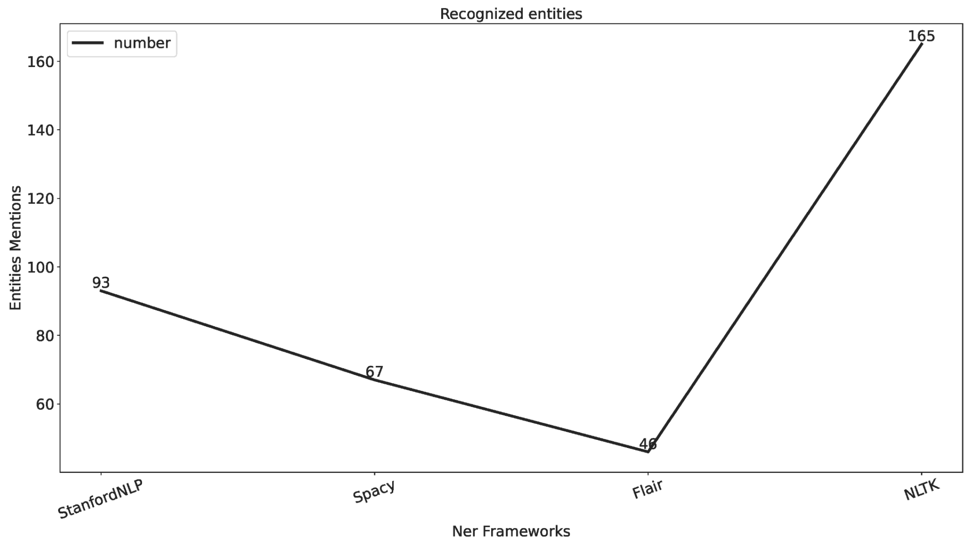
2.2. Text Extraction

For the text extraction part, the main task is to extract the text from the html documents that we obtained from the warc parser stage. In this part, what we mainly do is to filter the HTML tags and leave the text of the html.

To remove the label of the html and leave the text we want, we use BeautifulSoup4 to parse the HTML tree and extract the text content without the HTML tags. Before transmitting the string result to the Entity Recognition Phase, we cleaned the text by removing URLs, Hashtags and special characters (e.g. tabs) from the raw text using regular expressions.

2.3. Entity Recognition

To extract entity mentions in the text, we tried 4 named entity recognition frameworks which included StanfordNLP 4.2.0, spaCy v3.2, Flair 90.93 and NLTK. We took one HTML file text as a sample and then applied the 4 named entity recognition frameworks to the text. We then selected the framework which performed better. There are 1571 words in the sample, and after we tokenize them and apply the different NERs frameworks, we get the following results.



Although StanfordNLP and NLTK recognize more entities than spaCy and Flair, our final choice was between spaCy and Flair. After manually inspecting the results, we found there are some obvious mistakes of StanfordNLP and NLTK such as recognizing the first name and last name of one person as two persons. However, spaCy and Flair have no such issues. Also, when we look at the accuracy benchmark of spaCy and Flair on 2 corpus(ONTONOTES and CONLL ‘03) from the spaCy official documentation (https://spacy.io/usage/facts-figures#benchmarks), we found they behave almost the same on two corpus. Other than that, during the execution, the processing time of Flair is much longer than spaCy, hence we decide to choose spaCy as our NER framework.

2.4. Entity Linking

Entity linking is divided into two phases: First we obtain the candidates for one entity. This will result in many candidates per entity found. Then for each set of candidates, we rank them and select the best candidate as the final entity.

2.4.1 Obtaining Candidates

To obtain candidates, we use ElasticSearch to query the candidate entities' label and Wikidata Entity ID by setting the recognized entities as input. Candidates of the results of ElasticSearch are displayed based on a text similarity score ranking given by the ElasticSearch engine. Then, we pass these candidates to the next step to disambiguate and choose the best one. We used a local ElasticSearch engine using the docker image provided by the course professor.

2.4.2 Candidates Ranking and Selection

To select a proper entity we must rank the candidates obtained from the previous step. Therefore, we need ranking criteria. We used two ranking criteria: 1) Cosine Similarity of the Word Embeddings Vectors and 2) Entities Popularity.

First, we obtain the **Word Embeddings Vectors** from the entities inside the context in which they appear, and the vector from the isolated term label obtained from ElasticSearch. spaCy implements word embeddings vectors natively when analyzing any piece of text. Next we compute the cosine similarity between these two vectors and filter out all the vectors with a similarity lower than 0.80. The remaining entities are subject to the next two ranking criteria. This threshold was obtained by trial and error using the provided sample WARC file and sample output annotations.

Furthermore, if we find **exact matches** of labels between the found entity and the elastic search label, we then prioritize these exact matches and discard other candidates. All the exact matches will then go to the next step of disambiguation. An exact match is two words in which each letter of each position is equal.

For our second ranking criteria we compute the **entity's popularity**. In order to do this, we use the Wikidata knowledge base (KB) and SPARQL to query the KB. We defined the popularity of an entity as the number of triplets this entity has in the KB. We hypothesize that a more popular entity will have more triplets than a less popular entity. Each triplet found gives a +1 to the candidate. Queries to Wikidata KB were done through the **Virtuoso** server provided by the course professor.

Other tried techniques: We tried validating spaCy entity types output with Wikidata relations (e.g. PERSONS spaCy entities must be an **instanceOf** Human in wikidata). However, entities such as **Flash Player** being recognized as **PERSONS** were diminishing the accuracy of our implementation. Hence, we decided not to use this approach. We also tried seeking for the **entity relationships with the entities already found** in the context and using this as a third ranking criteria, however our results were not positively impacted by this technique.

2.5. Output Write

After having our best ranked entities selected we generate an output file. This file is in a tab separated values (tsv) format with three columns. The first column indicates the webpage with ID (e.g. “clueweb12-0000tw-00-00000”). The second column contains the entity mentioned in the text (e.g. "Flash Player”). The third column corresponds to the Wikidata Entity ID (e.g. “<http://www.wikidata.org/entity/Q857177>”).

3. Code Execution

To run our code, follow these steps:

A. Upgrade pip: `pip3 install --upgrade pip`
B. Install the needed dependencies: `pip install -r requirements.txt`
C. Start elastic search local server `sh es/start_elasticsearch_server.sh`. Wait a few seconds until elastic search is started.
D. Run the code: Two alternatives
    a. **SH File (Entity Linker + Score):** You can run the code by running the .sh file which assumes the following paths for input and output respectively: `data/sample.warc.gz` `sample_predictions.tsv`. And for the scoring input and output: `data/sample_annotations.tsv` `sample_predictions.tsv`
    b. **Separately:** You can also run the code by running `python3 main.py data/sample.warc.gz`. The latter parameter is the input of the WARC file. Then run `python3 score.py data/sample_annotations.tsv sample_predictions.tsv`. The latter parameters are the expected samples and the output file of the *main.py* program respectively.
E. **NOTES:** 1) On the first run, the program will download about 800MB of data corresponding to the spaCy models. 2) The program needs at least 1GB of free memory to load the spaCy models. 3) At runtime, the program will print on screen each of the entities found in the following format: wikidataID, entity label, ES score, ES entity label. This is not the file output, it is only a runtime output to give feedback to the user.

4. Concurrency

Our program tries to scalate by using multiprocessing when disambiguating entities. It disambiguates 30 parallel entities (at most) at a time. The latter parameter is configurable.

5. Limitations

After inspection of our results we found out that spaCy is not good at detecting religious entities (e.g. God, Holy Spirit). In addition to this, we did not find a way to discard non-existing entities (e.g. Non-famous humans who are not in wikipedia) other than the cosine similarity filtering. Finally, our execution speed is not optimal due to the use of Virtuoso external server.