Large Scale Entity Linking

Dhayalan Balakrishnan, Iuliia Parfenova, Rohit Shaw, Siying Zhang

I. Introduction

Since the invention of the World Wide Web, the amount of data on the Web has been growing exponentially. Billions of web pages, mostly in the form of text (unstructured data), have made it the largest data and knowledge repository in the world. However, plain text is ambiguous with respect to named entities which can have multiple meanings and could possibly change the meaning of the original text. This leads us to the need of processing and disambiguating web data on a large scale, given the growing size of the Web.

In this assignment, we perform Large Scale Entity Linking on a given collection of web pages.

II. Technologies Used

Our solution for this assignment uses WARC documents to perform entity linking. We chose Python 3.4 for this task as it was the latest available version on the DAS-4 cluster and also because it offers a rich set of libraries for data analysis; in our case - processing and analysis of large number of HTML documents.

However, a set of external libraries was needed to implement our solution and get the desired results. The complete list of Python packages that should be installed in order to run our program is

- pyspark
- nltk
- requests

For performing Entity Linking in particular, we used

- **NLTK** for removing stopwords and NER,
- **Elastic Search** queries for entity extraction,
- **Sparql** requests to Freebase RDF table for getting report tuples for the entities,
- **Spark** (without Hadoop) to perform all of the above in parallel for each web page.

III. Methodology

In this section, we describe the main work-flow we had adopted in the attainment of our goal, step by step. The constant baseline objective was precision-based entity linking, one phase after another.

A. *Basic Functionality*

1) Run a **process_page** function on Spark on the WARC archive.
2) Parse a WARC record and return the payload (WarcRecord class).
3) Parse the payload to words or pairs of words (see III-B) with regular expression (TextExtractor class).
4) Remove stopwords from the obtained list of words with NLTK.
5) Extract NERs from the payload, using NLTK's ne_chunk, and traversing the resulting tree.
   Output: list of all the named entities from the payload text.
6) For each word W_i extract its entities $\{FbId_{W_i}\}$ with Freebase query.
7) For each Freebase entity, extract its tuples from Freebase RDF with Sparql and put values of returned objects into sets, i.e. the structure of result is the following:
   $$W_N : \{\{FbId_{W_N} : \{FbRDFObject_{FbId_{W_N}}\}\}\}$$
8) The connectiveness of a pair of entities is defined as the size of intersection of sets of the entities related to them from RDF (Sparql). The best entity is then chosen as the one that has greatest connectiveness with some other entity from the context.
9) It was planned to use weighted combination of values of maximal connectiveness for each word in the context with higher coefficients being given to words positioned closer to the analyzed one, but this could not be performed once the cluster went down.

B. *Details on implementation choices*

- We used the following heuristics to improve recall and time of our approach:
  - During surface entity forms extraction from the document's text, if some HTML tag contained only two words, they were considered as one whole entity.
  - We also tried calculating links for words in some intervals around the considered word (± 10 words or ± 5 words), which improved the time significantly, but we couldn't measure its effect on recall (on one side, the smaller the interval, the smaller the recall should be; on the other side, inclusion of the whole text could add a lot of noise to the result) due to cluster issues.
- Due to frequently recurring problems with the cluster and Spark, we implemented mocks for pyspark (*pyspark_mock* file which can be included in *main.py* instead of the usual Spark), Elastic Search and Sparql (they generate random entity IDs) for convenience of local debugging and increase in development speed.

IV. PERFORMANCE

Measure	Score
F1	0.054
Precision	0.035
Recall	0.126
Runtime	Unavailable

** Scores very old. Performance of the latest version couldn't be checked due to cluster issues.*

V. CONCLUSION

During the first half of the development process, we were focused on polishing an approach that would work with the combination of Spark, Elastic Search, and Sparql for providing better performance based on scalability and running time per page. We had set our program to make only a single query to Elastic Search and only get one RDF tuple for each extracted entity. For this configuration, we received a recall of only 12.6%. Cluster issues prevented us from performing further experiments to improve results.

The ratio of entities predicted by our solution to the number of gold entities was 0.26 since we had to measure our results on an incomplete "silver" standard file, which significantly affected our precision (3.5%).

We had several ideas on how we could improve the precision, for example, by comparing our surface entity forms with labels returned by Elastic Search and considering only those that were similar enough. But we couldn't implement this improvement as we were unable to connect to the cluster.

VI. THE TEAM

Team Member	Contribution
D. Balakrishnan	NLP, NER, text parsing, performance improvement
I. Parfenova	Utils for ES, Sparql, WARC; Mocks for Spark, ES, Sparql
R. Shaw	Performance testing, Report writing
S. Zhang	F1 score calculation, NLP