Web Data Processing Systems
Project Report 2018 / 2019

Arumoy Shome *Vrije Universiteit*

Riccardo Bianchi *Vrije Universiteit*

Saba Amiri *Vrije Universiteit*

Shruti Rao *Vrije Universiteit*

1 Stage 1: Pre-Preprocessing

The input data is the form of WARC files so *warcio*, a Python library is used to read data [1]. This library is chosen because it was specifically designed for fast, low-level access to web archival content, oriented around a stream of WARC records. Taking advantage of this feature, *warcio* is used to stream the datafile [1]. *warcio* looks for the WARC records of type `response` with content type `text/html`.

For cleaning *html*, it was decided that *Beautiful Soup* would be used [2]. Beautiful Soup (BS4) was chosen due to its inbulit `get_text` method. *Beautiful Soup* comes with several parsers and we decided to use *lxml*. *lxml* is very fast and performant as it is written in C and can quickly parse the large amounts of text.

There were several re-occuring patterns of unwanted characters in the text and so custom Regex [3] was used to remove them. While Regex is not the most efficient, it is very good at specific tasks due to the ability to be as customized as required. First, *html* comments and their enclosed text, redundant, irrelevant characters and blank spaces were removed. Apostrophes were not removed however as they could be useful in providing context during the tokenization phase. To deal with carriage returns, a list of the universally allowed sequences were made and used to remove them from the text. Further, it was seen that *CSS* `<style>` tags were embedded as comments inside `<p>` tags. By adding certain filters, these *CSS* styles were removed.

Despite all the above, it was noted that *CSS* and *JavaScript* code were directly pasted in the text itself. Removing these would require very, very specific Regex that could not be generalized.

2 Stage 2: Preprocessing

For preprocessing, we actually implemented several steps including tokenization(both sentence and word level), lemmatization, stop-word removal, conversion to lowercase, NER and PoST. In the end, because of the method we used for entity linking, we only kept the tokenization and NER steps. We left the stop-words untouched in the text since removing them actually decreased the accuracy of NER in our tests. Also conversion to lowercase was discarded since our doc2vec model was trained in a case-sensitive manner. Our NER approach makes use of Stanford Core NLP [4] to perform the tagging. Since we decided to use Python as the sole language of our code base, we leveraged on PyCorenNLP [5]. This in one of the many libraries that provide APIs wrappers for the Core NLP Java Server implementation. The NER pipeline is composed by the following steps:

1. Start a local Stanford Core NLP Server that will provide the sentence annotation APIs.
2. Load the pre-preprocessed data of the WARC archives and split it into sentences with a fixed lenght of 128 characters (spaces included). This is done in order to reduce the payload of the NLP server requests.
3. For each sentence:
   (a) Execute a NER API call to the NLP server.
   (b) Process the resulting JSON and extract the annotated tokens.
   (c) Filter out all the tokens that have not been recognized by the tagger (ner = 'O').
   (d) Compose a list of all the surface words from the token with a valid NER tag.
4. Save the list of recognized entities into a CSV file for the entity linking phase.

3 Stage 3: Entity Linking

For entity linking, we used the doc2vec embedding model. We initially considered using a word2vec model and using the sentence around each named entity as the context, but it didn't yield very good results in our tests since the information about some entities were scarce in the document and related named entities were much further than a few words from our target named entity. The word2vec model was trained on English Wikipedia 2015 data dump in continuous bag or words mode. The model resides on HDFS as a binary file which the pipeline refers to for calculating the doc2vec embeddings. The calls to the model are done with the learning rate parameter set to 0.01 and the number of epochs set to 1000. Our pipeline for entity linking goes through the following steps:

1. Run the original document through word2vec and get its embedding.
2. Run the named entity through Elastic Search and get the first 100 relevant entities. Sort them by score and choose the top 10.
3. Run each entity through SPARQL and get the relevant abstracts.
4. Run each abstract through doc2vec to get the representing doc2vec embedding.
5. Calculate the cosine similarity of the original document vector with each of abstract vectors, then calculate the average distance over all abstracts related to each of our top 30 Elastic Search entities.
6. Choose the entity with the highest similarity average as our candidate entity

4 Extension: Machine Learning and Entity Retrieval

NLTK, a popular Python library for NLP contains a functional yet limited module `sem.relextract`. Extracting relations between two entities needs to be done using regex which needs to be written manually. Determining the type of relation that we are interested in is only possible when we have a good understanding of what our data looks like. However, this is not ideal when we are still exploring the dataset. In such a scenario, we want to first be able to scan for the types of relations that are present in the dataset.

PATTY provides a text document containing a list of patterns along with the corresponding relation. Using this dataset, we can train a model to predict the relation (if any is present) when given the filler text between two entities.

To generate a balanced training set, we looked at the number of patterns for each relation and those within the 75^{th} percentile was used. Further, 6000 patterns for each relation were chosen so as to have the same amount of training data for each type of relation.

Vectors generated from N-grams, word count, number of characters and average words were used as features. Using PCA, the total number of features were reduced by half (1000 to 500). Initially, 3 classifiers namely Linear SVM, Multinomial Naive Bayes and Random Forest Classifier were trained and Linear SVM was noted to perform the best. Lastly, the optimal parameter for the classifier were discovered using an extensive grid search.

The classifier was trained using a 5 fold cross validation and a score of 42.5% was obtained. Detailed code and experiments can be found on Github.

To improve the results either do LSA to generate topics for the short text or use a Python library like *libshorttext* for feature extraction. Using deep neural nets to do the classification is also an option that can be explored.

References

[1] Webrecorder. *webrecorder/warcio*. Nov. 2018.

[2] *Beautiful Soup Documentation*.

[3] *Regex*.

[4] *Stanford Core NLP*.

[5] *PyCoreNLP*.