Information Extraction of Seminars

Brendan Hart

December 5, 2016

1 Introduction

This report will discuss the ideas used to extract information from seminar data. The goal of the system was to extract the speaker, location, start time and end time of the seminar. As well as this, it was to construct an ontology, grouping the seminars into their respective topics or subtopics. This required a variety of techniques, with many similarities to Assignment 1, NER in the WSJ. As a result, this system involves using NERTagger from that assignment, but with slightly editted grammar rules and extra processing to make sure, for example, days of the motnhs are not tagged as speakers or locations. A download via `nltk.download` may be required to use NERTagger.
To use this system, a class named SeminarTagger is capable of tagging seminars in a given directory with given file names. There is also the OntologyMaker class, which is able to create an ontology from a dictionary of seminar objects. Seminar objects are instances of the Seminar class, which stores information about a given seminar. These all can be accessed by importing the file `seminartagger.py`.

2 Seminar Tagging

2.1 Preparing seminar data for tagging

The first step of the `tag` method was to split the seminar data into its header and abstract components, allowing for the data from the header to be easily extracted and for the abstract to be easily searched and considered seperately. The abstract is then processed by the NER tagger, which essentially performs chunking on the abstract[1]. This allows the ability to search through named entities in each sentence which is useful for extracting the location or speaker.

2.2 Extracting information

With the seminar data was in a more useful and accessible form, the next step was to extract the relevant information.

2.2.1 Sentences and Paragraphs

Paragraphs in the data seemed to be seperated by two new lines. As a result, to preserve the location of these new lines and to know which sentences should be grouped as paragraphs, the abstract is split on `\n\n`. For each element in the spit, it is replaced by the result of `nltk.sent_tokenize` with the element as the argument to the call. This not only gives where paragraphs should be as a result of the original split, but `nltk.sent_tokenize`[2] splits this further into sentences.
The system loops through each paragraph split, also looping through each sentence within that paragrph. At the start of each paragraph, an opening paragraph tag is added. Following this, it is necessary to determine whether a sentence produced by `nltk.sent_tokenize` should be tagged as a sentence in the text or not. For example, lines with just " SEMINAR " should not be tagged as a sentence. The system therefore contains rules to prevent against this when processing the sentences. Once the system is satisfied that the sentence is indeed a sentence, it encloses it with sentence tags and adds it to the new text, otherwise it is simply added to the new text without modification. Once the end of an element from the original split has been reached, a closing paragraph tag and the new lines are added to the text.

2.2.2 Time

The header always contains "Time:", which indicates at least one important time for the seminar. If two times are found in the header file, they are generally the start time and the end time in that order. If only one is found, it is likely to be the start time. The system finds all times given by "Time:" in the header using a regular expression, which was `(\d{1,2}(:\d{1,2}(\s*[AaPp]\.?[Mm])?|\s*[AaPp]\.?[Mm]))`[3]. It matches any time, such as "3pm" or "3:00". The amount of times found is then considered. If the amount of different times found is two, then the start time is assigned as the first and the end time is assigned as the second. If there is only one time found, then it is assigned as the start time.

Providing there was not an end time in the header, it is then necessary to look in each sentence of the abstract. Since end times generally occur after start times, the system proceeds to look for a time in the abstract which is the same as the start time in the header. To do this, it uses a method called `compareLooseTime` which will match times such as 3:00PM to 3pm. If it finds the start time, it then continues to look for the next time in the sentence, which it assumes to be the end time if found. In the event that the end time is continues for each sentence in the abstract.
Once the start time and, if it exists, the end time has been determined, the times can then be replaced throughout the seminar. To do this, a method called `looseTimeReplace` will find all times which match the start time or end time roughly in a similar to `compareLooseTime` checks for equivalence This returns a pair of lists with one list for the start times and one for the end times. The start times are then looped through and replaced in the text with the "stime" tag enclosing them, and similarly for end times but with "etime" enclosing them.

2.2.3 Speaker and Location

The speaker of a seminar and the location follow similar detection methods. Initially, the header is checked for the respective header tags, which is "Who" for "PERSON"-tagged named entities and "Place" for "LOCATION"-tagged named entities. If this is not found, the next step for the speaker is then to check in the abstract for any sentences which contain a named entity of the correct type and certain words. To extract named entities and their classification, the NERTagger class is used. First, the abstract is processed via the `process` method. Then, the named entities are extracted and flattened. Each flattened named entity is now considered in the text. For example, for location, if a named entity of type "PERSON" is found, and there is "giving" in the same sentence, the named entity is probably the speaker of the seminar, so it is tagged as such. If no person is found as a result of this, and "Who" is a key in the header, Then the first phrase is taken as the speaker. If this fails, then the first named entity tagged as "PERSON" in the text is taken as the speaker. Locations follow a slightly different path. If no relevant named entities are found in the "Place" key of the header and it consists of just one line, then the "Place" in the header is taken as the location. Failing this, the abstract is searched but this time for the label "LOCATION" and words such as "where". If still no location is found, the next two fallbacks are to the first found location in the abstract and then to the first line of the "Place" of the header if it exists.
Once the location and speaker has been found, instances of them in the seminar data are replaced by their tagged versions, such as `<speaker>John Smith</speaker>`.

3 Ontology Construction

For the construction of the ontology, the "Type" key from the header was considered. For most of the seminars, they followed a pattern of "foo.category.subcategory". This made grouping of them into a dictionary of dictionaries, which stored the seminar ID, fairly simple. Some of the seminars types were not this simple, however, and required some further processing. Those which followed the previously metioned pattern were split on fullstops. The subcategory was assigned to the last element in the split and the category to the second last. This was then inserted into the dictionary of dictionaries, so it looked similar to:
`'category': {'subcategory': ["id"]}`
For more complicated types, such as "AI Seminar", they are first converted to lowercase, and then the word "seminar" is removed as well as any common words such as "the". The type is now split on " ", and each split is looped through to see it is already contained in the subcategories. If it is, it is then appended to the list of IDs for the found subcategory inside the category. Ideally, this results in a dictionary of dictionaries which group all the seminars together based on their type.
Once an ontology has been created with the `OntologyMaker` class, the instance of the class can then have the method `search` called upon it, which allows a user to search for a given topic. The seminars found related to this topic are then printed out to the screen.

4 Evaluation

Since the system used `NERTagger`, the Naive Bayes classifier requried training. The classifier was trained on data from the WSJ. After this,

three tests were carried out in an attempt to determine the accuracy of the system. These were whether it classifies start time and end time correctly, whether it tags the speaker correctly, and whether it tags the location correctly.

Times	Speaker	Location
96.20%	52.71%	63.24%

The tests were repeated without the Naive Bayes classifier, so the system produced the following results with just DBpedia and handwritten rules.

Times	Speaker	Location
96.20%	37.50%	63.78%

It might be worth considering that the percentages may actually be higher than the test shows due to how the tests are carried out, "Mr. John Smith" does not match "John Smith", and will mean a fail even though they are the same speaker. A similar situation occurs with "Wean 5409" and "Wean Hall 5409".
From the above tables, it is evident that the Naive Bayes classifier provided a significant increase in performance for detecting the speaker, but does not provide the same benefit for location. The difference between location accuracy is so narrow that the margin of error from the test could be the cause of this. As a result, the DBpedia with Naive Bayes classifier managed to outperform the system without the Naive Bayes. This may be due to the classifier picking up on certain words which commonly follow a speaker or location.
There is evidently room for improvement with the speaker and location detection, which may be improved by using a classifier which has been trained on the seminar files as this will be more suited to the type of data which is expected. However, the system does an excellent job of detecting times and is close to perfection.
The ontology created from the list of seminars does manage to group seminars of the same topic together to some degree of success. It may be useful to consider how effective it would be to calculate the overlap between seminars which are grouped together. This would then provide a collection of words that are associated with that particular group. When it comes to classifying a seminar, it would be possible to then compare the group's pre-existing overlap to the most common words in the seminar, and group the seminar with the group that has the most common words. It would be useful to remove irrelevant words such as "the" from the common words, as these add little value to the classification.

References

[1] 7. Extracting information from text (2014) Available at: http://www.nltk.org/book/ch07.html (Accessed: 5 December 2016).

[2] (2016) nltk.tokenize package. Available at: http://www.nltk.org/api/nltk.tokenize.html

[3] 6.2. re - Regular expression operations (no date) Available at: https://docs.python.org/3/library/re.html (Accessed: 5 December 2016).