

Entity tagging of seminar announcements

Csongor Barabasi – 1636980

1. **Outlining the Task**

The task was, given 300 tagged emails, to build a tagger which will be able to tag 300 untagged emails accurately, in the same way the tagged emails were tagged. The corpus was tagged specifically with stime, etime, sentence, paragraph, location and speaker tags.

1. **Data Pre-Processing**

The first step was reading the corpus. This was done using *WordListCorpusReader* from *NLTK*. Then, using the *raw* property of the corpus reader, we could retrieve all the emails, concatenated. Before splitting it into email chunks, a cleaning process was done, where sequences of two or more characters like *‘=, ~, %, -, \_, \*,* |*’* were removed. Next step involved splitting this long text of 300 emails into emails chunks, and then split up each email into a header and a body part, which was then stored in an *Email* object. Finding each email was done with the regular expression

split the text according to each email’s header. Then, by splitting each email according to the word *‘Abstract’*, I was able to split each email into a header and body part. The result of the pre-processing part was a list of *Email* objects.

1. **Sentence and Paragraph Tagging**

In the given corpus paragraphs can be identified as being separated by two newline characters. Hence, I could obtain a list of paragraphs from each email body by splitting the text as:

Sentence tagging was as simple as paragraph tagging. Using *NLTK’s* built-in *sent\_tokenize* function, which uses an instance of *PunktSentenceTokenizer*, I was able to split the text into sentences and tag them accordingly.

1. **Time Tagging**

First, I extract the line from the header which gives information about time using the regex

Then we extract all the existing timestamps from this line using the regular expression

Next, I checked whether there were one or two timestamps found. If there was only one, it was tagged as *stime*, if there were two than the first was tagged *stime* and the second as *etime*. Next, I scanned through the body of the email and looked for all the occurences of the times found in the header, and tagged them in the body accordingly (eg. if there was one timestamp in the header, that was tagged with *stime*, so all occurences of that timestamp in the body were tagged as *stime*).

1. **Speaker Tagging**

As a first attempt, I check whether there exists a line starting with the word *‘Who:’* in the header. In case it exists, I tag the content after the semicolon as speaker and then I scan through the body and tag each occurrence of that speaker’s name with the speaker tag.

In case there is no information about the location in the header, first I split the body of the email into sentences using the *‘sent\_tokenize’* function of *NLTK.* Then each sentence is split into words with *NLTK’s ‘word\_tokenize’* function, and then POS tagging (described at point 7.1) is applied. Next I apply a NER tagger (described at point 7.2) which will return an entity-tagged sentence in *nltk.Tree* format. From that tree, I extract all the entities tagged as *‘per (=PERSON)’* and tag all their occurrences in the body. This approach is guaranteed to tag every person’s name in the body, so the recall of the algorithm is higher than its precision.

1. **Location Tagging**
2. **Custom Algorithms**

The reason why I decided to not use the in-built version of the following algorithms is because in my opinion nowadays sophisticated Machine Learning algorithms can outperform the old-fashioned NLP algorithms, and also the task was to build our algorithms as much as possible, to understand the core of NLP.

* 1. **POS Tagger**

As a POS tagger, I trained a Decision Tree Classifier using *scikit-learn*, which after learning from already annotated data, was able to POS tag new sentences. I used the *‘treebank’* corpus as data for my classifier, because it has a huge amount of data already tagged. Instead of just relying on features like the previous word (bigram) or the two previous words (trigram), after some research on the internet, I found a few relevant word features based on which my classifier should work, eg:

* is it the first word in the sentence?
* is it the last word in the sentence?
* is the first letter capital?
* is it all capitalized?
* prefixes
* suffixes
* previous word
* next word
* is it numeric?
* has hyphen?

After gathering my training data and defining the features my classifier relies on, I had to prepare my data for training. For this reason I split the pre-tagged corpus into two arrays: the input, containing dictionaries of features about each word, in order as they appeared in the corpus; the output, which contained the expected POS tag for each word. Once the training was done, I was able to test it, since I split my corpus into 80%-20%, and the resulting accuracy was **94.5%**.

Because the training took about 8 hours, I had to save it to disk, and whenever an instance of the *POSTagger* class is created for use, the model loads up from disk. As input it accepts an array of words (word tokenized sentence) and outputs an array of tuples of the form *(word, tag)*.

* 1. **NER Tagger**

My first attempt was to train a NER tagger using the in-built *NLTK* tools, but there were a few issues:

* the training corpus was too big, and the program was running out of memory, but from my perspective, in most of the cases more data means better accuracy
* even though the accuracy was 93%, I believed that I could achieve better results
* based on my POS tagger experience, I thought I could come up with better feature sets that better describe the data

For these reasons I decided to use Machine Learning again, to build a NER tagger. When choosing my model, I had to choose one which supports **Out-Of-Core Learning**. This is a learning process through which we keep the training data out of RAM. We rather load it in batches and do **Incremental Learning** with the classifier. I chose the Perceptron model, because it trains fast and also gives good results. As a dataset I used the *‘Groningen Meaning Bank’* corpus, which is already NER tagged. After pre-processing the data by separating the words from the resulting NER tags, I had to get documented on word features which could improve my classifier, and came up with:

* lemma of word
* is it capitalized?
* is it lowercase?
* is it a number?
* ending dot?
* next word and details about it
* previous word and details about it

and so on…

I split my corpus again, into a 80-20 ratio and trained it. In the end the classifier was saved to disk, and whenever needed it can be loaded from file using *‘pickle’*.

The accuracy achieved was much better than the first NER algorithm’s, resulting in **97%**.