Named Entity Recognition in the WSJ Corpus

George Taylor

Semester 1 - 2016/17

Abstract

The aim of Named Entity Recognition (NER) in this case is to recognise entities that have been referenced in the text and determine the type of the entity from (Person, Location or Organization). The program will be provided with a set of tagged and untagged articles from the WSJ corpus as well as a set of test articles to analyse the preformance of the system.

Extracting Training Data

The first task is to extract the already tagged entities from the set of tagged documents which is done using the requlars expression:

```
<ENAMEX TYPE=".*?">.*?</ENAMEX>
```

which returns a list of all tags in the documents from which it is trivial to extract the entities and their types.

Once we have the entity within the tags we can part of speech (POS) tag it using `nltk.tokenize()`, giving us the POS tags for every entity (Bird et al., 2009).

Grammar Creation

The next step of the process is to take these entities and create a grammar that can be used in a chunker (Bird et al., 2009) to extract unseen entities. Firstly, any occurences of ")", "(" or ":" are removed as these are reserved characters for the grammar and would cause issues while keeping them would have no forseeable benefit. A list of tuples is then created that gives the POS tags and entity type, e.g.

```
[(["NNP", "NNP"], "PERSON")]
```

which are then compiled into a dictionary that gives the frequecy of each tag sequence for each type. From this, the program will immediately remove any tuples that occur below a threshold frequency (currently set to the average frequency multiplied by 2.6). The tags are then sorted using a heuristic that favours their length and then their frequency, thus promoting greedy evaluation. The resulting grammar will be in this format:

```
ORGANIZATION: {<NNP><NNP><CC><NNP><NNP>}
ORGANIZATION: {<NNP><NNP><IN><NNP>}
PERSON: {<NNP><NNP><NNP><NNP>}
ORGANIZATION: {<NNP><NNP><NNP><NNP>}
PERSON: {<NNP><NN><NNP>}
ORGANIZATION: {<NNP><NNP><NNPS>}
ORGANIZATION: {<JJ><NNP><NNP>}
LOCATION: {<NNP><NNP><NNP>}
ORGANIZATION: {<NNP><IN><NNP>}
PERSON: {<NNP><NNP>}
ORGANIZATION: {<NNP><NNP>}
PERSON: {<NNP>}
LOCATION: {<NN>}
PERSON: {<NN>}
ORGANIZATION: {<NN>}
LOCATION: {<NNP>}
ORGANIZATION: {<NNP>}
```

While this grammar will distinguish between different types of entity, it is not accurate enough to be used in reality. For example, because of the presence of rules such as `PERSON: <NNP>` many unwanted nouns will be matched and we can not simply assume that these are all names. For this reason, the grammar is only used to extract entities and the types of these entities are ignored.

Extra Datasets

Throughout the program a couple of extra datasets have been used:

- DBpedia: This was orgininally a downloaded dataset that had been reformatted giving a list of entities and their types. However this has been modified to now use the SPARQL query language to access the entities and their types directly from the online database. Because of the time it takes to ocnduct these online queries this is only done as a last resort if the entity could not be related any other way. The advantage of this method is that the database will be continuously updated so new entities can be identified without having to download any extra data.

- Names corpus: The nltk package provides a corpus of names, this makes the task of matching names significantly easier as using the corpus alongside some other rules can lead to extremely accurate name matching.

- IE-ER corpus: The "Information Extraction and Entity Recognition" corpus provided by the Information Technology Laboratory division of the NIST provides a corpus of 1506 entities with types Person, Location or Organization, so while small it means that up to 1506 entities can be related almost instantly.

Chunking

Using the grammar that has now been created, the program can extract entities from the untagged data. By iterating over the files we can get the contents for each file one at a time. The program then uses `nltk.sent_tokenize()` then `nltk.word_tokenize()` to split the text into words. Using `nltk.pos_tag()` and `nltk.RegexpParser(grammar)` we can get a parse tree of the text that can be iterated over in order to extract the entities. These entities can now be processed individually in order to determine their type. In addition to this, the `get_relation()` function takes a list of the ten previously identified entities which are then used to help identify organisations (e.g. if *'Hercules Inc.'* has been identified as an organisation then the next few referrences to *'Hercules'* are likely referring to the same entity). The function also takes the previous word in the sentence, this is used to quickly identify relations as if the entity is preceeded by *'in'* then it is most likely a Location.

Entity Relations

Because relating entities has the potential to be extremely time consuming the first step is to apply some of the more obvious rules to identify the type of the entity. The program will first check the corpora to check if it can immediately identify the relation. It will then check the previous word to see if the entity can be related from this. Following this there are three functions that attempt to identify the entity as a specific type (executed in this order):

- `is_organization()`: This will first check if the entity contains any of the words used solely for businesses (*'Inc.', 'Corp.', 'PLC', etc.*). Failing this, if the entity is in full capitals, has a length less than 7 and is only one word then it is likely to be an abbreviation for a company (this can cause issues with over-generation as it will match entitys like *'GDP'* that are obviously not companies). If this fails then it will go through the previous ten entities checking if the current entity can be found within any of them.

- `is_name()`: This will first check if the entity contains any of the words commonly associated with names, e.g.*'Mr.'*, *'Mrs.'*, *'Jr'*, *'etc'*. If none of these can be found then it will check if the entity contains at least one word that can be found in the names corpus, and the rest of the words are either initials (a single capital letter followed by a period) or start with an uppercase letter.

- `is_location()`: Because most locations will have been matched in the initial stages of getting the relation, most locations that are left over are abbreviations of a location such as *'Del.'* or *'Tex.'*. For this reason the program will return a location if the entity is a single word with at most 7 letters, the first being a capital and the last being a period.

If all of these options are exhausted then the program can attempt to access DBpedia in order to determine the entity. The program will query DBpedia using the SPARQL (Prud'hommeaux and Seaborne, 2013) query:

```
SELECT ?t
WHERE {
OPTIONAL {
<http://DBpedia.org/resource/%s> a ?t
} .
}
```

where `%s` is the name of the identifier. This query will return the DBpedia listing for the type of the entity if it can be found. It is then trivial to check if the listing contains any of the required relations.

Observations

As previously mentioned, the grammar is created by taking only those rules that have frequency greater than the average multiplied by 2.6. Testing the program on the test data with different values for the multiplier yields the following results (these results were calculated before extensions to the datasets were made):

Multiplier	Precision (%)	Recall (%)	F_1
0	33.63	41.4	37.11
0.4	63.56	59.71	61.57
0.8	63.69	61.88	62.77
1.2	63.77	63.16	63.46
1.6	67.32	70.68	68.96
2.6	69.3	75.83	72.42
4.0	68.99	76.66	72.62

It is also worth noting that for smaller values of the multiplier, the larger resulting grammar causes the NER to run significantly slower. From this table it is evident that larger values for the multiplier produce better results, however larger values also lead to a considerably longer runtime as a result of a greater dependence on online resources.

Results

The process for completing the named entity recognition is as follows:

1. Load the entities from the tagged training data.

2. Create a grammar from these entities.

3. Optionally test the grammar on the untagged training data and gather statistics.

4. Test the grammar on the untagged test data.

5. Load the entities from the tagged test data and calculate statistics for the NER performance.

When testing on the untagged training data, the program can produce a precision of 73.02% and recall of 83.16%. However because the program stores the entities and relations loaded from the tagged training data, this is a somewhat inflated result. The entities in the untagged data are common to those loaded from the tagged data giving it an inherant advantage. When completing NER on the test data the program can still achieve a precision of 69.3% and recall of 75.83%. While this is noticably lower it is still sufficient to conclude that the program is successful when being run on unseen data.

References

Steven Bird, Ewan Klein, and Edward Loper. Natural language processing with python, 2009. URL `http://www.nltk.org/book/ch07.html`. Accessed: 29/11/2016.

DBpedia. Accessing the dbpedia data set over the web, 2016. URL `http://wiki.dbpedia.org/OnlineAccess`. Accessed: 21/11/2016.

Eric Prud'hommeaux and Andy Seaborne. Sparql query language for rdf, 2013. URL `https://www.w3.org/TR/rdf-sparql-query/`. Accessed: 26/11/2016.