NLP assignment evaluation

Part 1: entity tagging

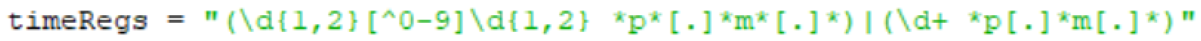
My code is contained in the assignment.py file, as the accompanying files do not need to be run. They were used to generate pickled files or to test the main code.

assignment.py follows a fairly simple structure, working through each email in turn. The emails are first split into a header and a body (by finding the 'Abstract:' string), and useful information is extracted from them before the tagging begins.

I used a variety of techniques to tag the entities in each email, from RegEx to POS tagging to WSD. Below, I will explain each entity in turn, my techniques for tagging them and the overall results (as shown in evalOutput.txt).

Start time

This was by far the simplest entity to tag, as there is a 'time' line in the headers of every email. After using a series of Regular Expressions (see below) to find the times, I used rudimentary analysis to determine which was the start and end times, and accounted for different formats of time (i.e. with or without AM/PM, in the 24-hour and 12-hour formats. This approach was very effective to increase my program's recall, achieving 91%.



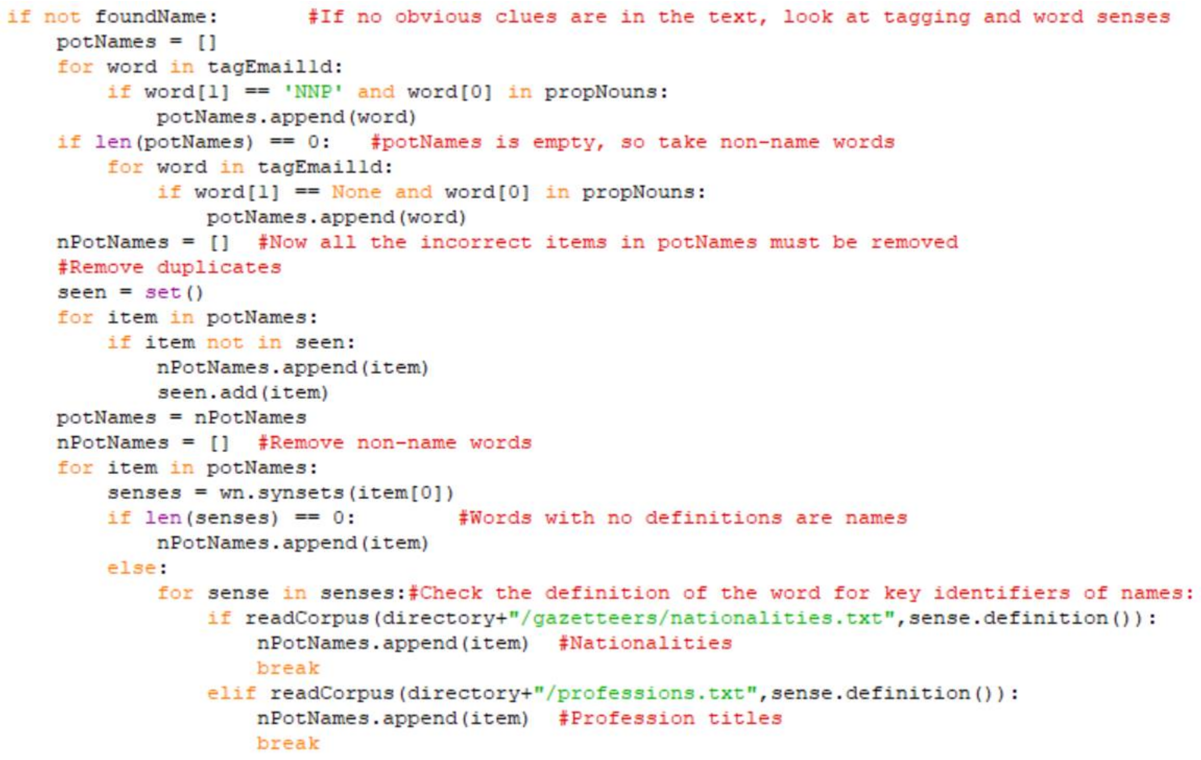
```
timeRegs = "(\d{1,2}[^0-9]\d{1,2} *p*[.]*m*[.]*)|(\d+ *p[.]*m[.]*)"
```

However, the program's precision is quite low at 27%. This is mostly due to times appearing in different formats throughout the email, such as '1:00 PM' being included in the header but a '1pm' being used in the body. My program would pick up the first but not the second, a problem which I could fix given more time with the program.

Speaker

The speaker was by far the hardest bit of information to extract, as the provided corpus missed a lot of names and incorrectly categorised many others. Therefore, I had to use many layers of validation before a name can be presented. My program searches first the header, then the body for names, tagging them with a trained POS tagger before using corpora and the senses of words to attempt to whittle the list down to one name. Currently, the program checks the definition of potential names against both a corpus of nationalities (contained in the 'Gazetteer' file) and one of professions (professions.txt), as I noticed that both show up heavily in the definitions of name (as most name definitions are talking about a person). Common regular expressions are also searched for, such as 'Visitor' and 'Speaker'.

You can see an excerpt of my code below, where the corpora are used to remove words that were identified as names by the tagger whilst not actually being meant to be used as names (such as 'Tuesday').

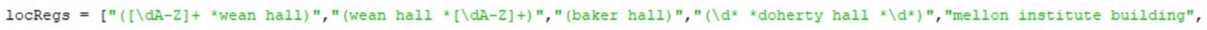


```python
if not foundName:       #If no obvious clues are in the text, look at tagging and word senses
    potNames = []
    for word in tagEmailld:
        if word[1] == 'NNP' and word[0] in propNouns:
            potNames.append(word)
    if len(potNames) == 0:   #potNames is empty, so take non-name words
        for word in tagEmailld:
            if word[1] == None and word[0] in propNouns:
                potNames.append(word)
    nPotNames = []  #Now all the incorrect items in potNames must be removed
    #Remove duplicates
    seen = set()
    for item in potNames:
        if item not in seen:
            nPotNames.append(item)
            seen.add(item)
    potNames = nPotNames
    nPotNames = []  #Remove non-name words
    for item in potNames:
        senses = wn.synsets(item[0])
        if len(senses) == 0:        #Words with no definitions are names
            nPotNames.append(item)
        else:
            for sense in senses:#Check the definition of the word for key identifiers of names:
                if readCorpus(directory+"/gazetteers/nationalities.txt",sense.definition()):
                    nPotNames.append(item)  #Nationalities
                    break
                elif readCorpus(directory+"/professions.txt",sense.definition()):
                    nPotNames.append(item)  #Profession titles
                    break
```

This process is incomplete, as I had to focus on other areas of the project, but I am sure that with more time I could greatly improve the stats of 35% recall and 27% precision. Some immediate ideas that occur to me are expanding the names corpora to include names in foreign languages, and using the senses of words around each name to find the correct one.

Location

To create the location tagger, I first searched through all the emails to find the 83 that contained a 'who' section in their header. I then used the results of these headers to form a list of Regular Expressions that between them generated 73% recall (you can see some of these RegExs below). The location tagger suffered similar problems to the time tagger, as it can account for different abbreviations of locations but not if they are included in the same email. For example, 'Wean 6234' and 'WeH 6234' are both allowed by my tagger, but it will only pick up one at a time, meaning emails that use both will be tagged incorrectly. As with the other tags, I believe given more time I would be able to greatly improve the precision score of 42%.



```python
locRegs = ["([\dA-Z]+ *wean hall)","(wean hall *[\dA-Z]+)","(baker hall)","(\d* *doherty hall *\d*)","mellon institute building",
```

Sentence/Paragraph

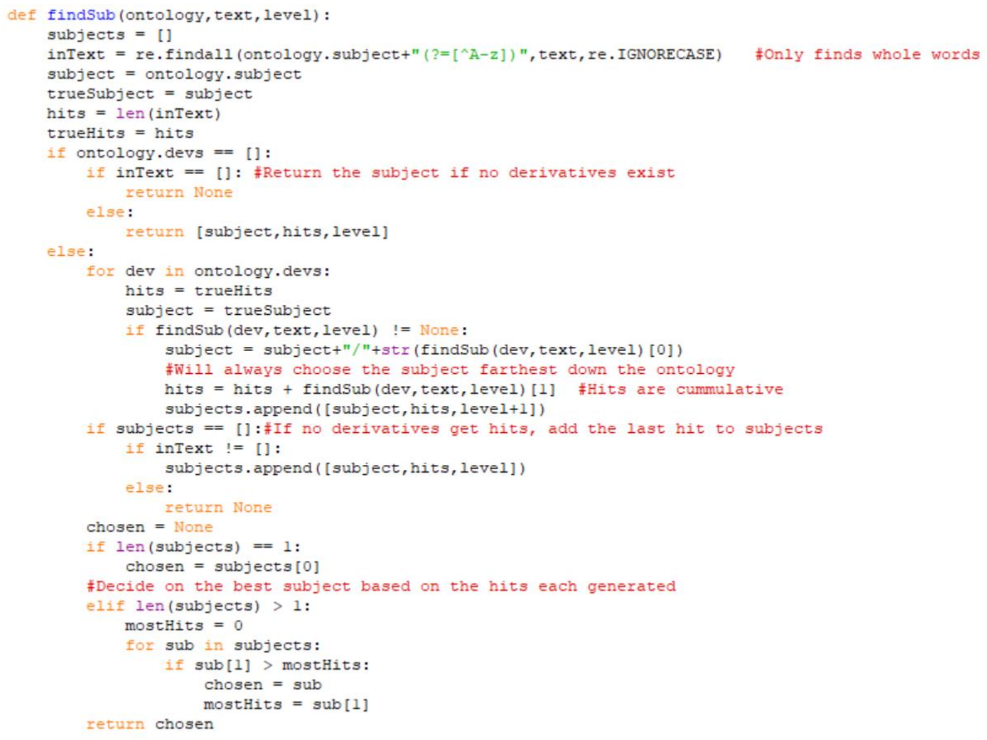
As I had focused most of my time on the name tagger, these two taggers were implemented fairly rudimentarily. They both rely on a simple Regular Expression, sentences find a sentence-ending punctuation and a space before words continue, whilst paragraphs simply look for two adjacent newline characters. The statistics below are the result, although some of the unfortunate precision statistics are due to the nature of the tagging (the test data tags sentences in different places to my code).

Tag	Precision	Recall
Sentence	1%	11%
Paragraph	1%	33%

Part 2: ontology construction

I initially attempted to use wikification to generate the ontology, but this proved too difficult due to the lack of a coherent API for wikification in Python. Instead, I processed the HTML script of a large Wikipedia document (https://en.wikipedia.org/wiki/List_of_academic_fields) showing the different academic disciplines, and formed this into a complete ontology.

My ontology forms a complete hierarchy that is used to correctly identify the academic department in the email. Recursively travelling down the hierarchy, my program uses regular expressions to find each subject in turn (making sure that they form a whole word and not just parts of it). Once a match is found, this is added to a list of possible matches before the program continues, ensuring that the match deepest into the ontology is used. A string is then constructed that shows the 'path' of the chosen subject, which is then added to the output file. This process is applied first to the header, then the body, as the header is more likely to produce a correct match. You can see this function below.



```
def findSub(ontology,text,level):
    subjects = []
    inText = re.findall(ontology.subject+"(?=[^A-z])",text,re.IGNORECASE)   #Only finds whole words
    subject = ontology.subject
    trueSubject = subject
    hits = len(inText)
    trueHits = hits
    if ontology.devs == []:
        if inText == []: #Return the subject if no derivatives exist
            return None
        else:
            return [subject,hits,level]
    else:
        for dev in ontology.devs:
            hits = trueHits
            subject = trueSubject
            if findSub(dev,text,level) != None:
                subject = subject+"/"+str(findSub(dev,text,level)[0])
                #Will always choose the subject farthest down the ontology
                hits = hits + findSub(dev,text,level)[1]  #Hits are cummulative
                subjects.append([subject,hits,level+1])
        if subjects == []:#If no derivatives get hits, add the last hit to subjects
            if inText != []:
                subjects.append([subject,hits,level])
            else:
                return None
        chosen = None
        if len(subjects) == 1:
            chosen = subjects[0]
        #Decide on the best subject based on the hits each generated
        elif len(subjects) > 1:
            mostHits = 0
            for sub in subjects:
                if sub[1] > mostHits:
                    chosen = sub
                    mostHits = sub[1]
        return chosen
```

My program has shown unprecedented success, identifying a subject in 84% of the emails. As there is no test data, precision cannot be identified, however by observing the results myself I can confirm that the majority of the time the correct subject is identified, and most of the incorrect identifications are from emails with no clear subject. Given more time, correctly pruning the hierarchy would be the next step, as particular sections have far too broad titles (such as 'strategy' in the military science section).