Information Extraction of Seminars Report

Me:

Sophie Guile 1643360 sxg661@student.bham.ac.uk

Collaborated with:

Aikaterini Chelioti 1581991 axc591@student.bham.ac.uk

File Tagging

Sentence Tagging

To tag sentences, we used the english.pickle tokeniser to split the file data up into sentence tokens. Of course not every token should have been tagged as a sentence, so at this point we checked for certain sequences of characters, such as two spaces in a row or two * characters etc, that tend to appear a lot the seminar files, but very rarely if never in sentences. Quit a lot of time there are lines in the code such as “TOPIC: Electron Diffraction”, which I didn’t want tagged, so we checked the sentences against a regular expression to make sure that they were’t in this format:

```
illegalRegEx = re.compile("[a-zA-Z]+:.*")
```

The problem with this method, is that while the sentence identification code doesn’t have any major problems, the tokeniser itself would often give us chunks like this:

```
                    GASCHNIG/OAKLEY MEMORIAL LECTURE

                        WAR <speaker>STORIES FROM</speaker> ANDREW

                              James H. Morris
                        Professor of Computer Science

                          Wednesday, <speaker>April</speaker> 8, 1992
                                Time 3:30pm
                              Wean Hall 7500
   It  has been five years since I stopped working on <speaker>Andrew</speaker>, so the
freedom
   of information act -- along with the passing from power of several people
   --  allows  me  to  tell  some  candid stories about it. In 1983, the ITC
   started and grew to about 30 people. Eight were IBMers, ten  or  so  were
   newly  minted  Ph.D.s  from  our  very  own  department. We <speaker>built
<speaker>Andrew</speaker></speaker>,
   partially in the model of Spice and the Alto  system.  It  achieved  some
   real  successes  but  didn't accomplish all that we'd hoped.  <speaker>Much</speaker>
of the
   problem was intrinsic to the <speaker>way</speaker> the world is, compared to how we
thought
   it   was.   I   often   meet   people   who   still   have   some   of   my   1982
   misperceptions.
```

This chunk obviously has sentences within it, but you can see that none of them have been tagged, because the tokeniser just picked it up as one massive chunk. As a result the program saw a lot of white space and decided that this whole chunk is not a sentence, but missed the sentences inside. I think that maybe after we got my chunks, we should have

checked for sentences within the chunk, either by using a regular expression, or potentially another tokeniser.

Here are the scores for Sentence Tagging:

```
-------------------------------------------
sentence
-------------------------------------------
Precision   :   0.0
Recall      :   0.0
F1 Score    :   0
-------------------------------------------
```

As you can see, these scores could literally not possibly have been any worse. However, I think the reason for this was because the sentence tags in our tagged files were in subtly different place than in the test files, for example with 305.txt:

Our file:

```
<paragraph><sentence><speaker>Much</speaker> was learned in this fast-paced, high risk
research project in
areas ranging from robot configuration to telepresence control to
field operations and logistics.</sentence>  <sentence>One measure of the success of the
```

Test file:

```
<paragraph><sentence>Much was learned in this fast-paced, high risk research project in
areas ranging from robot configuration to telepresence control to
field operations and logistics</sentence>.  <sentence>One measure of the success of the
```

(When it checks if the sentences are the same, it removes all tags e.g. the <speaker> tags around the word “Much” first, so they will not affect the scores)

Our code has pretty much found the correct sentence. The only thing making it was incorrectly tagged was the position of the full stop; in our file the full stop is before the </sentence> tag, but in the test file it was after the </sentence> tag. Therefore to optimise our code in this case, all we have to do is make sure that when we are tagging sentences, we trim the full stop off the end of the sentence before tagging it. I have looked at other files, and this seems to be the case with them to.

It is also possible that the results may have been affected by new lines, because our tagged files have many more new lines than the test tagged files. This is due to them keeping the old new lines, and adding in new ones when needed.

I actually did decide to put both my theories to the test by running the testing code removing new lines and full stops from the sentences when I found them. Removing new lines had no effect on our scores, but removing full stops gave us very significant improvement:

```
-------------------------------------------
sentence
-------------------------------------------
Precision   :   0.52
Recall      :   0.47150259067357514
F1 Score    :   0.4945652173913044
-------------------------------------------
```

This means that by not taking into account the position of full stops, our code has captured around half the sentences. It is still obviously not an amazing result, because it misses a lot and captures a lot of fake sentences, but it does show that we're going in the right direction.

Paragraph Tagging

We did paragraph tagging after sentence tagging. This is because by knowing where the sentences were, it was easy to find groups of them together. To do this, we used a very simple regular expression:

```
paragraphRegEx = re.compile("<sentence>[^¾]*?</sentence>\n\n")
```

The reason we put "[^¾]*" is because we tried ".*", but there were many characters that were not included in the ".", so we just took everything except for a character that was very unlikely to come up, and this seemed to work. I know that there are probably better ways to do this, but this was sufficient for what we were trying to do. The question mark after "[^¾]*" is there because we don't want the regular expression to be greedy in it's matching; we want it to capture the smallest chunk of sentences possible, so we don't have paragraph tags around all the paragraphs in a file rather than round each individual paragraph.

This paragraphing tagging worked part of the time, but was far from perfect. The main area where it failed was in the cases where a paragraph did not end in a double new line, for example:

```
<paragraph><sentence>Revision Rules were first introduced by A.</sentence>
<sentence><speaker>Gupta</speaker> (1982) and N. <speaker>Belnap</speaker>
(1982) as tools in the theory of truth, and have found their most
detailed exposition to date in <speaker>Gupta</speaker> & <speaker>Belnap</speaker>
(<speaker>1993</speaker>), where they
provide the foundations for a general theory of (possibly circular)
definitions.</sentence>  <sentence>Revision Rules are non-monotonic inductive operators
that are iterated into the transfinite beginning with some given
"bootstrapper" or "initial guess."</sentence> <sentence>Since their iteration need not
give rise to an increasing sequence, Revision Rules require a
particular kind of operation of "passage to the limit:" rather
than, as is usual in the <speaker>monotone case</speaker>, taking the cumulative result
or union of what is obtained at previous stages, in Revision theory
we take the "inferior limit" of the sequence, i.e., the set of
items that do not oscillate cofinally in the sequence, but eventually
stabilize.</sentence>

<sentence>This idea is applied in recursion theory to obtain a recursive
operator G(x,phi) whose iteration over omega2 beginning with
any total function satisfying certain relatively simple conditions
gives rise to sets of increasing arithmetical complexity.</sentence></paragraph>
```

The top paragraph actually ends in " \n\n" (space double new line), rather than "\n\n". We could have altered the regular expression to capture this too, and any other over variations that we could have found. For example, if our full stops were in the correct place, we'd

have to allow ".\n\n" and ". \n\n" etc. There's also the possibility that a paragraph end may end at the end of a file, which we'd have to account for.

Here are our scores for paragraph matching:

```
-------------------------------------------
paragraph
-------------------------------------------
Precision   :   0.25
Recall      :   0.13043478260869565
F1 Score    :   0.1714285714285714
-------------------------------------------
```

(The paragraph scores were not affected by the position of the full stop, because I removed the sentence tags before calculating scores.)

Our paragraph scores are very low, and this is mainly due to the missing of sentences in our sentence tagging (around 50%), and sentences ending in things other than two new lines.

Named Entity Tagging

The named entity tagged consisted of two main stages: finding the named entities in the text, and classifying these entities as speakers or locations.

Finding the named entities:

To find the named entities in the text, we first POS tagged the text. Because we were trying to find *named* entities, we used a tagger that read in the names corpus, and then used our trigram, bigram, unigram and default taggers as back off taggers in that order.

After POS tagging the text, we defined a grammar that would take combinations of tags that indicate a named entity, for example a sequence of proper nouns. However, one thing that we did notice was that even with the names taggers, there are lots of things that we considered to be proper nouns that were only tagged as nouns. In attempt to solve this problem, we changed the grammar to include nouns as well, but this gave us the opposite problem where we were picking up lots of things that were not a named entity.

In the end, we decided to use use two grammars: a strict grammar and a broad grammar. The strict grammar would only pick up things that we were certain were named entities and generally only looked at proper nouns, and then any extra entities were picked up by the broad grammar. Everything picked up by the strict grammar would be classified as either a speaker or a location, but things from the broad grammar would have the possibility of being unclassified.

Classifying named entities:

To classify the named entities, we decided to use wikification. Firstly we wrote two files, one where we go through the training data, extract all the locations, got the relevant words from wikipedia and wrote them all to a file, and the same from speakers. Then, when we were attempting to classify and entity we went to wikipedia and got all the words relating to

this specific entity, then compared them to the words in our two files and saw which ones were a closest match.

To do this we used a naive bayes classifier, which look at each word we found for the entity we were attempted to classify and looked at how often the word came up for locations and speakers, and added to the score accordingly.

We didn't rely on naive bayes alone though. Before attempting naive bayes we would look at the entity and see if it was a direct match to a speaker or location. To do this, we had two files, one with the speakers and one with the locations from the training files, and then we just looked to see if the entity was inside this.

If we were classifying an entity from the broad grammar, if the entity was one of the files, we would automatically classify it, if not it would go to the bayes classifier. If the difference between scores for location and speaker were 50 or greater, we would classifying it as it's best matched tag, otherwise we would say that the entity was unclassified and not tag it.

So overall for our classification, our method was:

(1) Extract entities using the strict grammar.
(2) Extract entities using the broad grammar and only store the ones which have not been extracting with the strict grammar.

Then for each entity

(1) Check if it is a 100% match to any of the training entities, if it is, classify it accordingly.
(2) If not, calculate the best match tag, and the difference between scores for the two tags.
(3) If this entity was found with the strict grammar, classify it as its best match tag.
(4) If this entity was found with the broad grammar, only classify it as its best match tag if the difference between scores in greater than or equal to 50.

Here are our scores for named entity tagging:

```
-----------------------------------------
location
-----------------------------------------
Precision    :    0.5263157894736842
Recall       :    0.14492753623188406
F1 Score     :    0.2272727272727273
-----------------------------------------
-----------------------------------------
speaker
-----------------------------------------
Precision    :    0.08681672025723473
Recall       :    0.2755102040816326
F1 Score     :    0.13202933985330073
-----------------------------------------
```

The reason that the precision for speaker if so low, but is much higher for location, is because my naive bayes classifying was very biased towards speakers. If it was a speaker, then it would be classified as a speaker, but lots of other irrelevant things were classified as a speaker too:

```
                    VASC Seminar

 Time:          Monday, <speaker>March</speaker> 16, 1992, 3:30-<stime>5:00 pm</stime>
 Place:         Wean Hall 7220
 Speaker:       Josef Skrzypek
                Machine Perception Laboratory
                University of California at Los Angeles
                Los Angeles <speaker>CA</speaker>, USA

 Comment:       If you are interested in meeting with <speaker>Josef Skrzyek</speaker>
                please <speaker>contact Patty Mackiewicz via email</speaker> at
patty@ius5.

 Title:         Neural net models of lightness and <speaker>color constancy</speaker>.
```

You can also see that the location was missed in this file, which happened quite often. If something was tagged as a location however, it was often actually a location (or at least part of one, which was not picked up by our scores). This also explains why the recall was higher for speakers than locations.

In order to get rid of a lot of the nonsense that was picked up by the broad grammar, I wrote a function to check for nouns/ “proper nouns” (e.g. the word “will”, because it’s also a name) within an entity , and if they began with a lower case it ditched the whole entity automatically. I probably could have though of other types of strings that would be found by mistake, e.g just titles on their own, and got rid of these too.

I also feel like my classifier was biased, because there were more speakers than locations in the training data. Therefore it was much more likely to classify something as a speaker than a location, because the prior probability of something being a speaker is much higher (in my code I did take prior probabilities into account). I think that in that case, it is almost alright that the classifier in biased, because it was very rare that a location was classified as a speaker.

Overall, I think that the main problem with the classification was not the naive bayes classifier, though it could be improved upon with more training data, but the actual extracting of the named entities. To optimise our code by refining the grammar, identifying fake named entities that are extracted sooner and maybe using regular expressions on the data to try and pick up anything that was missed by the grammar. The naive bayes classifier was also relatively inefficient, because it had to go to wikipedia for every single entity, but getting rid of incorrectly extracted entities before attempting babes would definitely speed up the time of tagging.

Time Tagging:

To do time tagging, we first went and found every time in the text, regardless of whether or not it was an end time or start time, using this regular expression:

```
time = re.compile('[0-9][0-9]?\:[0-9][0-9][ ][am|AM|pm|PM|a.m|A.M|p.m|P.M]{2,3}')
```

After this, we gave all of the times we found the tag name of “stime”.

Then we used a very similar regular expression to go through and find the end times, by only finding the times that game after a dash:

```
time_end = re.compile('[ ][-][ ][0-9][0-9]?\:[0-9][0-9][ ][am|AM|pm|PM|a.m|A.M|p.m|P.M]{2,3}')
```

Then we went back to the times we found before and overwrote ever single one we found here with the “etime” tag.

Here are our scores for time tagging:

```
-------------------------------------------
etime
-------------------------------------------
Precision    :   0.9230769230769231
Recall       :   0.5
F1 Score     :   0.6486486486486487
-------------------------------------------

-------------------------------------------
stime
-------------------------------------------
Precision    :   0.7297297297297297
Recall       :   0.40298507462686567
F1 Score     :   0.5192307692307693
-------------------------------------------
```

Overall our time scores are relatively good. I think that the reason that quite a few times were missed, is because we didn’t allow for there to be no am or pm after the time. Mainly though, the times that we did capture were the actual times. I think the slightly lower precision is mainly down to the fact that times in the main body of text were also tagged. Maybe it would have been better if we only looked for times outside of sentence tags, so avoid classifying any false times.