INFORMATION EXTRACTION

1. ENTITY TAGGING

The program takes untagged seminar announcements from the “untagged” folder and automatically labels the sentences, paragraphs, speakers, locations, start times and end times of the seminars.

The files, headers, contents and tagged will contain different contents of the emails.

A. Tagging the paragraphs

i. Split the contents of the emails by ”n\n” to get the paragraphs
ii. A paragraph will be tagged if there is a sentence in it.
iii. There are no paragraphs tagged in the headers of the emails.
iv. There are no paragraphs tagged in the contents of the emails if the paragraphs contain any of the following: 'type:', 'who:', 'topic:', 'dates:', 'time:', 'place:', 'duration:', 'host:', 'when:', 'where:', 'speaker:', 'title:', 'furtherdetails:'
v. After splitting the paragraphs by ‘\n\n’, I used nltk.sent_tokenize to tokenize the sentences. After that, going through all of the sentences in the paragrpaph, I check if they are sentences or not using the method is_sentence(sentence).
vi. If a sentence contains any of the following: :'type:', 'who:', 'topic:', 'dates:', 'time:', 'place:', 'duration:', 'host:', 'when:', 'where:', 'speaker:', 'title:', 'furtherdetails:', it is not tagged.
vii. To check if a sentence is actually a sentence, I used nltk.pos_tag on all of the words in the sentence to see if there is a verb. If a verb is found, the sentence is tagged
viii. Finally, if there is at least a sentence found in a paragraph, the paragraph is tagged.

B. Tagging the start and end time

i. When reading through the emails, I saw that, in the headers, the time is usually mentioned, but when it is only the start time mentioned, and if there is “Posted by:” statement that contains a time, that time could be considered the end time, which is wrong. So, I decided to tag the time in the headers separately from the time in the contents of the emails.
ii. To tag the times in the headers, I first check if there is a ‘Time:” statement. If there is then I check if there is a “Posted by:” statement. If both statements are found, I use the tag_time(text) method to tag the start and end time found in the text between ‘Time:” and “Posted by:”.
iii. To tag the time in the rest of the email, I just used the tag_time(text) method

iv. The tag_time(text) method uses the following RegEx '\b((0?[1-9]|1[012])([:][0-5][0-9])?(\s?[apAP][.]?[Mm])|([01]?[0-9]|2[0-3])([:][0-5][0-9]))\b' to find any time that matches the pattern. If nothing that matches it is found, nothing is tagged. If at least a time is found, it is tagged as the start time. If a second time is found, it is tagged as the end time. If there are any other times found, they are not tagged.

C. Tagging the speakers and the locations

i. To tag the speaker and location in a header, I am looking for "Who:" and "Place:" using RegEx. If any information is foundI tag it in the header and in all of the tagged emails.

ii. I use the same techniwue for tagging the speakers and locations in the contents, but this time looking for "WHO:", "SPEAKER:" or "WHERE:".

iii. I thought of using two lists all_speakers and all_locations and put in them all of the names and locations I find, and then for the emails that do not have them explicitly specified use the lists, but after the evaluation, the values were only increased by 0.5% for the speakers and somehow decreased for the locations, so I decided not to use them anymore.

D. Tagging all of the entities and writing the tagged emails

i. The map() method splits the contents of the emails in headers and contents, and maps the tagged emails in the tagged map.

ii. The write() method takes all the tagged emails and writes them as .txt files in the "tagged" folder

iii. The tag() method puts the read(), map() and write() methods together, and, when called, does the entity tagging task.

2. THE EVALUATOR

To compare the results of the tagger to the tagged test data from the "test_tagged" folder, I used an evaluator which compares the files and outputs the accuracy, precision, recall and F1 measures for all the tags. To do this, the evaluator looks for each tag in both "tagged" and "test_tagged" files, and compares the tagged text to calculate the true positives, false positives, and false negatives. Using those values, it then calculates the accuracy, precision, recall and F1 measures for each tag. These are the values of my code's evaluation:

TAG	Accuracy	Precision	Recall	F1 measure
total	52.56%	60.57%	79.89%	68.90%
sentence	44.71%	56.96%	67.53%	61.79%
paragraph	37.73%	45.22%	69.49%	54.79%
stime	88.00%	91.67%	95.65%	93.62%
etime	74.07%	87.43%	82.90%	85.11%
speaker	34.22%	37.87%	78.05%	51.00%
location	37.73%	45.22%	69.49%	54.79%

3. CONSTRUCTING THE ONTOLOGY

This part of the program classifies the emails by the subject or topic. I made it so that it classifies the emails related to Computer Science by the cs topic. When running the program, the user is asked what subject they're interested in. If they write "Logic", the output will be a list containing the document IDs of the emails that are related to Logic.

A. Creating the ontology tree

i. "tree" is the ontology tree, and it contains two subtrees, "Science" and "Art.

ii. The "Science" tree contains three subtrees, "Computer Science", "Chemistry" and "Physics".

iii. As this classifier only classifies CS emails, I only populated the "Computer Science" subtree, but the classification could be expanded to classify all of the subjects and their subclasses.

iv. To get the CS topics, I used the get_topics() method that searches using RegEx for the "Topic:" statement in the emails, extracts that information and puts it in the all_topics list. The contents of the list are then written in a txt file "All Topics" using the write_all_topics() method. I then read through all of them and manually deleted all of the irrelevant words, such as prepositions, irrelevant verbs or nouns, etc.

- Instead of doing this manually, I tried looking for the words in the topics that had the highest frequency, but soon realized the results were not very useful, so I decided to do this manually, which did not take too long.

After deleting the irrelevant information, I saved the new file as "Selected Topics". I then read the content of the file using the read_topics() method, tokenized the words and added them to the selected_topics list. I used this list to decide on the subclasses for "Computer Science", then stemmed all of the words, put the stemmed words in the stemmed_topics list and used them to find keywords for the CS subclasses. All of the keywords are in the keys tree.

v. To classify the CS emails, using a for loop, I go through all of the emails, look for the keywords in the contents, and if I find them, is_cs becomes True, so the email is about CS. Then, I look for keywords in the "Type:", "Topic:" and contents of the emails and classify the emails.

vi. The ontology method takes a parameter, which is the subject, and prints the document IDs of the emails that are related to that subject, or "This subject does not exist" if the subject is not in the tree. This method is called in the main method and takes user input.