David Eastman Research Narrative

This is a narrative description of what I've done so whoever comes next can see why I chose to do things the way I did.

The first thing I did was look at the C-Or-DiAL corpus files that Dr. Jeon provided. There was an XML file with the entire corpus contained. Having worked with XML in Python before, I knew there was a library called BeautifulSoup that could easily parse the XML and allow access to the raw text (i.e. not annotated). I wanted the raw text so that I could feed that into the FreeLing corpus analyzer to get the textual analysis.

(FYI, the BeautifulSoup library is why I chose Python to be my language of choice. There are Java XML parsers, as well, but since I’d worked with BeautifulSoup with XML most recently, I thought this would be a good option. The library and documentation can be found at <https://www.crummy.com/software/BeautifulSoup/> or searched for with Google).

The XML file wasn’t perfect, however. There were missing tags that BeautifulSoup complained about when trying to parse. Once I’d identified where the tags were missing (by seeing what lines the parser threw an error at), I noticed a pattern where one particular tag at the end of each “conversation” (called “texts”) in the corpus was not being closed properly. So, I manually inserted the tag at the places where the parser failed, thus cleaning the document. That cleaned corpus xml file is in the Github repository.

Once the corpus was cleaned, I created a small python script (process\_corpus.py) that uses BeautifulSoup and regular expressions to run through and output just the raw text of the corpus without the annotations and other extraneous characters. I named this file “raw\_output.txt”

Once I had the raw text, I needed to get access to the FreeLing analyzer. Going to the web page of the FreeLing project (<http://nlp.lsi.upc.edu/freeling/node/30>), I downloaded the Windows version “freeling-4.0-win64.zip” and unzipped it on my computer. The entire unzipped folder is 1.2 gb. In the /bin/ directory, there is a file called “analyzer.bat” that is meant as a sample analyzer (since FreeLing can be customized from source and compiled into whatever options you want). This sample analyzer file had enough options to get started, so I just used that one.

To run analyzer.bat, I used a shell environment (bash, specifically), so I could feed it the right input and get the output into a file. The bash command I used to get my raw text into the FreeLing analyzer was “./analyzer.bat -f es.cfg < raw\_output.txt > freeling\_output.txt”

* “./analyzer.bat” calls the executable
* “-f es.cfg” references the Spanish language configuration file that is found in \freeling-4.0-win64\freeling\data\config. The default configuration file seems to work fine, so I didn’t change it.
* “< raw\_output.txt” is what inputs my raw text that I’d created with my Python script.
* “> freeling\_output.txt” is what outputs the results of the FreeLing analyzer into my chosen filename.

The FreeLing output came in the format “word, root\_word, code/tag, confidence”. For example, “recientes reciente AQ0CP00 0.987805”. FreeLing basically compares each individual token with its dictionary and takes a guess at what the root word is and what the mood/tense/person/etc. is based on its “knowledge” of the Spanish language.

So, now that I had this output, I wanted to process it. So I created another Python script, (process\_freeling\_frequency.py) that takes in the freeling\_output.txt and counts up how many times a certain root word appeared and separates the different parts of speech into their own files. The results of this script are found in the “parts\_of\_speech\_frequencies” folder.

Next, we wanted to break down the parts of speech into even more detail. That is, break down the output not only into parts of speech (which is determined by the first letter of the code/tag), but also into the tenses/moods/person etc, (represented by each individual letter of the entire tag).

So, I created a Python script (process\_freeling\_breakdown.py) that takes the freeling\_output.txt and just outputs each word’s tag along with the count, like so: “AO0FP00 último 4”. This output is in “breakdown.txt”

Then, in order to parse each of those tags, I created a separate script (parse\_breakdown) that basically translated the cryptic tags into their English word equivalent, then output the results again. The codes of the tags that FreeLing outputs can be found on the website <https://talp-upc.gitbooks.io/freeling-user-manual/content/tagsets/tagset-es.html>.

Since this data is very detailed (meaning you can do things like count how many times the verb “ser” is use as a semiauxiliary verb in the subjunctive in the imperfect in the third person singular, for example), I decided to make the output of this script in table format to make it easier to navigate. So I output my results into comma-separated files (.csv), each starting with the prefix “breakdown\_”.

Since the data is in .csv files, which is a widely-accepted format for tables, this data can be used with any sort of analytical software that works with tables. As a proof-of-concept, I used the software RapidMiner, which is data mining software free for students, to do some statistical analysis of the verbs by their tenses. I chose RapidMiner because I’d already used it before, but there are many options here. In fact, Python itself has some libraries that I might use going forward, in order to not require someone to know how to use RapidMiner to continue this project. But it’s a fast way to get results if used correctly.

For instance, I used Rapid Miner to create a breakdown of the frequency of verbs by mood, tense, person and number and put the results in Analysis.xlsx. I decided to use Excel since that is a simple way for anyone to do basic statistical analysis. One can sort by the field in which they’re interested, or sum the percentages to see what percent of the entire corpus is comprised of whichever subset they want.

Above written October 3rd

I made two more analyses available in Analysis.xlsx, which were breakdowns according to the category of the conversation (clase, charlas, conversación, diálogo,entrevistas, conversación con finalidad predeterminada,conferencias, conversación en un lugar de trabajo) and according to key words (abogacía, arquitectura, artesanía, boda, cambio, carácter, carrera, casa, etc). In this way, it’s possible to see how the choice of verb tenses/moods/etc. was influenced by the type of conversation and the topics being discussed.

At this point it was desired that there be some way to highlight particular parts of speech \*in\* the corpus itself, so that it could be verified whether FreeLing’s output was accurate.

To do this, I created several scripts that used FreeLing’s output to “rebuild” the corpus from it. The first one I attempted was process\_corpus\_with\_metadata\_pos\_tagged.py. This was designed to match up FreeLing’s output with the original annotated corpus using Regular Expressions and working line by line, tagging each word with its FreeLing tag. This was so as to keep the annotations in the corpus. However, there were many complications keeping the annotations because FreeLing was never fed the annotations (and would ignore them in the output anyway), so trying to used regular expressions over the very idiosyncratic annotations proved too messy with too many edge cases.

So, a simpler approach was created in process\_corpus\_with\_metadata\_pos\_tagged\_no\_annotations.py. In this script, the FreeLing output itself was used to rebuild the conversations. Naturally, no annotations were included, but it was felt that the annotations were secondary to the purpose of checking the accuracy of FreeLing’s output. So in this script, each word in the FreeLing output was tagged (e.g. <w t="NCFS000">cosa </w>) then concatenated with the previous word. Since each of the conversations was processed separately, the metadata for each conversation was self-contained and could be included in the output CorpusCordial\_Tagged.xml file.

This XML file was then processed to become a nicely formatted HTML file (using process\_tagged\_corpus\_with\_metadata\_xml\_to\_html.py) so that it would be human-readable, and could be enhanced with javascript and css. A helper script (process\_tagged\_corpus\_with\_metadata\_html\_format.py) was used to automatically add some convenient html tags that would be usable by css and javascript). Truthfully, these two files could have been combined, but were instead used in succession.

After the HTML was set up automatically by the above scripts, I manually added dropdown boxes and various other HTML to make it possible to select parts of speech. Javascript and css were used to make it possible to highlight the chosen parts of speech. The entire HTML webpage (CorpusCordial\_Tagged\_HTML\_Formatted.html) along with javascript and css are in the corpus\_html folder.

The javascript file script.js handles the highlighting. A built-in function of css is to be able to select any HTML element with a certain tag. This is the basis of how the highlighting works. Essentially, after a user selects his/her preferred parts of speech (with the dropdowns changing based on verb/noun/adjective/etc), pressing “Highlight Words” builds a tag from those options, then the javascript goes and finds all the tags starting with that tag. An important feature of the implementation is the ability to handle wildcards (e.g. if a user wants to select all verbs in the 1st person, but doesn’t care what tense or mood). So, the approach I took was to first grab all of the possible words, then selectively remove the ones that didn’t match at each place in the tag, with “\*” indicating that the user doesn’t care, so ignore any mismatches there.

It was then desired that there be a frequency count according to the words chosen. This involved adding some javascript that would do a simple frequency count calculation for the words chosen by the highlight function, then build some simple HTML to display in the appropriate section of the page.

Above written October 31st