Rachel Rosenberg
MSiA 490: Text Analytics
October 5, 2019

Comparison of Two Libraries: NLTK and Spacy

The libraries I completed this analysis in were NLTK and SpaCy. I downloaded the "20 Newsgroups 18828" dataset, which as duplicates removed and contains only "TO" and "FROM" subject headers. I decided to download this version of the data because it was the cleanest; I did not want to spend extra time cleaning the data, removing duplicates, or removing TO and FROM tags. I completed all of the analysis on one file in the dataset: file 59846.txt from the "Sci.Space" library. This was a challenging dataset because it contained many numbers, equations, and non-UTF8 characters (Roman numerals, Greek alphabet, etc). To begin, for all datasets, I replaced numbers with "0" (I did not care about the numbers themselves), converted everything to lowercase, dropped blank words ('‘), and removed punctuation. I primarily cleaned with the String package in Python. All of these cleaning steps made it much easier to process and read the rest of the data.

The NLTK package had great functionality for everything I wanted to do. It was simple and quick to tokenize with the word and sentence tokenizer functions – the hardest part for me was keeping track of where breaks were in lists (for a while it was giving POS tags for every character, etc). Stemming and finding POS tags were also very straightforward; NLTK has great built-in functions for both that allow these functionalities in just a few lines of code.

Even with as easy as NLTK made it to munge the data, I prefer SpaCy for analysis. SpaCy is easy to use, it plays nicely with list comprehensions, it is very fast, and the syntax is intuitive. Another reason why I prefer SpaCy is that the POS tags it gave made much more sense; rather than gibberish letters like "CD" and "NN" for words, it gives human-readable tags like "VERB" and "ADV". However, I had an issue with my package installation, so it ended up being somewhat more difficult to get the tags out of the data than it was using the NLTK package.

The main difference between NLTK and SpaCy is the algorithm used; NLTK comes with a variety of algorithms, while SpaCy comes with only one state-of-the-art algorithm that is consistently updated as NLP technology improves. This makes SpaCy faster, but NLTK more versatile. The two languages also treat text differently; SpaCy uses a purely object-oriented approach (which matches Python well), while NLTK deals exclusively in strings. This is a style difference and does not significantly affect processing time; it is the user's choice which style they prefer.

Parallelization in NLTK took processing time from 35 seconds to 16. Using the same multiprocessing package and the exact same dataset, parallelization in SpaCy did not work initially; SpaCy has its own parallelization package and appears to run indefinitely if threading is attempted in the Multiprocessing package. SpaCy does this because it is designed to be used at scale. However, once I got it to work, multithreading took SpaCy's processing time from 33 seconds to 22 (on average). This is comparable to the results obtained using the same dataset in NLTK.

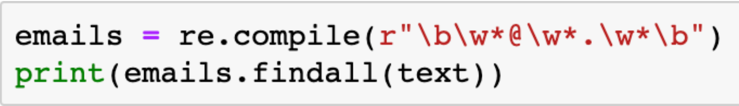
Rachel Rosenberg
MSiA 490: Text Analytics
October 5, 2019

Write and test regular expressions:

Text used: We are very excited for the Christmas party next month on 12-12-19. As such, we need to prepare food and decorations for the event. Debbie will be heading up food preparation; if you would like to be a part of her team, please contact debbie101@gmail.com by December 4 2019. Judy will be leading decorations for the event; if you would like to work with her, please contact judybee43@aol.net by 11/29/19. Alan will take the lead on the White Elephant portion of the event; if you would like to help Alan out, please reach out to alanjones@jones.jo by December 3, 2019. If you don't have time but would like to donate money to the event, please contact the studio at admin@namaskaryoga.com after November 30, 2019 and before 12-11-19 so that we have time to appropriately allocate funds between Debbie, Judy, and Alan. Thank you!

2.1 Match all emails in text and compile a set of all found email addresses.

Code:



```
emails = re.compile(r"\b\w*@\w*.\w*\b")
print(emails.findall(text))
```

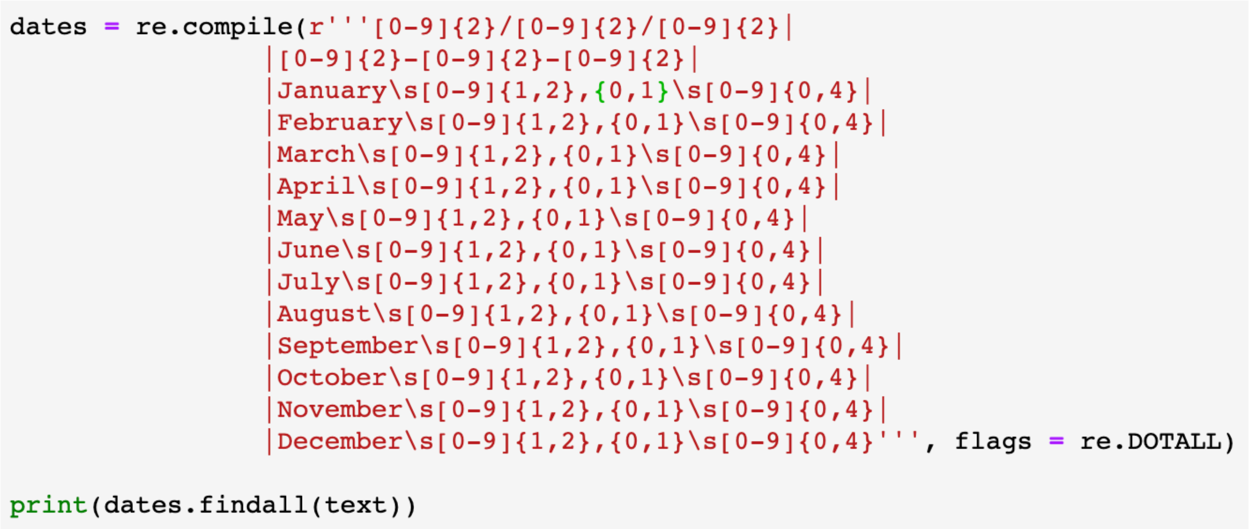
Answer:

```
['debbie101@gmail.com', 'judybee43@aol.net', 'alanjones@jones.jo', 'admin@
namaskaryoga.com']
```

Rachel Rosenberg
MSiA 490: Text Analytics
October 5, 2019

2.2 Find all dates in text (e.g. 04/12/2019, April 20th 2019, etc).

Code:



```
dates = re.compile(r'''[0-9]{2}/[0-9]{2}/[0-9]{2}|
                |[0-9]{2}-[0-9]{2}-[0-9]{2}|
                |January\s[0-9]{1,2},{0,1}\s[0-9]{0,4}|
                |February\s[0-9]{1,2},{0,1}\s[0-9]{0,4}|
                |March\s[0-9]{1,2},{0,1}\s[0-9]{0,4}|
                |April\s[0-9]{1,2},{0,1}\s[0-9]{0,4}|
                |May\s[0-9]{1,2},{0,1}\s[0-9]{0,4}|
                |June\s[0-9]{1,2},{0,1}\s[0-9]{0,4}|
                |July\s[0-9]{1,2},{0,1}\s[0-9]{0,4}|
                |August\s[0-9]{1,2},{0,1}\s[0-9]{0,4}|
                |September\s[0-9]{1,2},{0,1}\s[0-9]{0,4}|
                |October\s[0-9]{1,2},{0,1}\s[0-9]{0,4}|
                |November\s[0-9]{1,2},{0,1}\s[0-9]{0,4}|
                |December\s[0-9]{1,2},{0,1}\s[0-9]{0,4}''', flags = re.DOTALL)

print(dates.findall(text))
```

Answer:

```
['12-12-19', 'December 4 2019', '11/29/19', 'December 3, 2019', 'November
30, 2019', '12-11-19']
```

Rachel Rosenberg
MSiA 490: Text Analytics
October 5, 2019

Main ideas of Charniak paper:

A text parser is a process to turn input data (usually strings of text) into some machine-or-human-readable format. This can mean a variety of different things for the field of text analytics, as required functionality for a parser depends heavily on the content and structure of the text as well as the structure of data required for its application. Parsers often attempt to organize the text data by grammatical structure; that is, to attempt to capture the order and syntax of the text in a uniform, consistent, and reproducible way, regardless of what the text itself says. One way to structure parsed data is in a *syntax tree*; this requires *tokenizing* the text, or splitting it into individual words, and *part-of-speech (POS) tagging* the text, or marking which words are nouns, verbs, adverbs, and so on. Once words are tokenized and tagged, they are organized into a tree which preserves the sentence-like structure of the data and is easier to manipulate for later modeling. Depending on the size of the text data, this can be an extremely computationally intensive task. This is the goal of the Charniak paper: to quickly and accurately build syntax trees from large amounts of text data.

To assign POS tags, parsers generally assign a probability to each word of being each part of speech, and keep the POS tag with the highest probability for each word, grouped by sentence in the corpus. This creates a *parse*: a POS tag, structured, for each word in each sentence. Different parsers assign these probabilities in different ways. This model uses a combination of each word's POS tag, the lexical head (the word that determines a sentence or phrase's meaning: **dog** in the phrase "big red **dog**") of the sentence or phrase, and each word's relevant context. It also starts by guessing the head's "pre-terminal value", the word directly preceding the lexical head which determines its tense and context. This ends up being an important method, as this determines the direction that the tree for the sentence will branch. Thus, in each iterative labelling of sentences, the model is using current context, as well as three rounds of prior probabilities, on a set of *features*: functions that define patterns in the data.

This methodology creates a model that is accurate, reproducible, and flexible: it can extend to many sets of data, provided that new features are trained. This model is trained on the entire corpus of data, and is then smoothed, so that features which rely heavily on rare words are deleted. The "maximum-entropy-inspired" title of the model comes from this process of conditioning and smoothing the model, removing those features that are helping the least at each round of iteration. The phrase "maximum-entropy" is synonymous with flexibility and speed, choosing a distribution of probabilities in which "the selected distribution is the one that makes the least claim to being informed beyond the stated prior data, that is to say the one that admits the most ignorance beyond the stated prior data."

This maximum-entropy inspired parser is more accurate (has 13% error reduction) than other parsers in large part because it uses three prior probabilities for each parsed solution to create its labels. It is also key that it starts by guessing the head's pre-terminal value in each sentence; this practice alone increases model accuracy by as much as 2%, because it strengthens parse probabilities significantly by adding context to key words. This all results in a new maximum-entropy inspired parser that is fairly simple, highly flexible, and highly accurate, making it a great parsing option to use for the processing of large amounts of data.

Additional resources used:

https://en.wikipedia.org/wiki/Principle_of_maximum_entropy

https://www.aclweb.org/anthology/P97-1003.pdf

http://www.cse.unsw.edu.au/~billw/nlpdict.html#headfeature

https://www.asc.ohio-state.edu/demarneffe.1/LING5050/material/structured.html