Qiao Yu (Finn Qiao)
Text Analytics HW 1

1.

Three text processing libraries were used for comparison. NLTK, Textblob, and Spacy.

The corpus was a dataframe of food reviews on Amazon that was downloaded as a dataframe and aggregated as a single corpus by concatenating all the full review text in the final column.

Using a sample of 100,000 characters, the comparisons in the following categories can be made:

1. Speed
    a. For the purposes of tokenizing, NLTK surprisingly came out as the fastest library by achieving the task in around half the time of the other 2 libraries. This advantage does not necessarily scale linearly and Spacy becomes more efficient with larger corpi and especially when it comes to the context of productionalizing. According to research, Spacy is definitely the most optimized in the backend and is the choice for larger scale applications.
    b. It was interesting that while NLTK performed much faster for tokenization, Textblob and Spacy both vastly outperform NLTK in slightly more computational actions like stemming and pos tagging.
    c. The most basic function of Spacy of creating an nlp object with text could take a little time to run but once it is run, almost all other results like lemmatization, pos tags, start of sentence flags etc. all come with the resulting object. This vastly speeds up the process if multiple operations were required on the text.
2. Learning curve
    a. NLTK provides the most gentle introduction to text preprocessing by having individual pre packaged functions to import and simply use. All of NLTK functions work on the specific thing that you call it on. For example, there is a specific function you call for each form of stemming (PorterStemmer and SnowballStemmer) and you can import each as a specific function.
    b. Textblob and Spacy are much more object oriented in which a “blob” is created for Textblob and you can slice and dice the object by words, by sentences etc. The resulting Word object for example, can have its own class of functions called on it. As a result, the learning curve for Textblob and Spacy are a lot steeper but the steeper learning curve comes with increased performance and modularity (described below).
3. Modularity
    a. NLTK modules are right out of the box and there isn’t much customization available or involved. You use the functions as they are and it is very difficult to try to modify any source code to achieve what you need.
    b. This is absolutely not the case for Spacy in which there is an API that one can call to create customized objects for each use case. For example, a simple tokenizer in Spacy can catch multiple different types of specified suffixes, prefixes, separators etc. The high degree of customizability ensures that no matter the

use case or format of the text, a custom object can be easily created to fit the required purpose.

2.1

```
In [230]: # 2.1 Match all emails in text and compile a set of all found email addresses.

          email_re = re.compile(r'[a-zA-Z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}')
          x = email_re.findall('dd@dd.com not_an email email.email.com t3hisisanemail@dac.com')

In [231]: x

Out[231]: ['dd@dd.com', 't3hisisanemail@dac.com']
```

2.2

```
In [322]: # 2.2 Find all dates in text (e.g. 04/12/2019, April 20th 2019, etc).

          dates_re = re.compile(r'((0?[1-9])|(1[1|2])|jan[a-z]*|feb[a-z]*|mar[a-z]*|apr[a-z]*|may[a-z]*|jun[a-z]*|jul[a-z]*
                                |aug[a-z]*|sep[a-z]*|oct[a-z]*|nov[a-z]*|dec[a-z]*)[\/\.\-](?:[0-2]?[1-9]|3[0-1])[\/\.\-]
                                  ((?:19|20)[0-9]{2})', re.I)

In [323]: test_dates = '01/1/2019 January/2/2019 dec-12-2020 3/2/2013 13/23/2018 10/23/2018'

In [330]: print ([x.group(0) for x in dates_re.finditer(test_dates)])

          ['01/1/2019', 'January/2/2019', 'dec-12-2020', '3/2/2013', '3/23/2018']
```

3.
Charniak Paper:

In machine learning, a parser for natural language processing refers to the process of breaking sentences down into individual constituents that correspond to the part of the sentence they form and represent. Sentences follow specific rules around syntax and structure that can be broken down into different simpler components such as words, grammatical use, and phrases. Fully breaking down a full sentence into individual words can at times, be too granular and not capture useful information. For example, interpreting “the red fox” as one phrase is much more useful and simpler than trying to interpret “the”, “red”, and “fox” as three individual words.

With the new approach, Charniak is attempting to improve the accuracy of parsing sentences into individual constituents. The test and train sets for this approach are derived from the Penn tree-bank style parse tree in which the breaking down of individual sentences can be represented in a tree format. The individual leaves of the trees represent individual words and the leaves are branched together to represent a block that can be a phrase like a noun or verb chunk.

In trying to maximize the accuracy of such a task, a set of features is created for each word position that can include many different features for information such as the positional tag of the word (whether it’s a noun or adverb etc.), words before and after, and the lexical head (the word that is most determinant of the syntactic nature of the phrase (“fox” determines that “the

red fox" is a phrase for a noun). By understanding the likely order of different syntactic categories and words within these phrases, probabilities can be determined for whether a word is within a phrase or part of another tree or in a branch of its own. Ultimately the model is optimized by considering these sets of features and maximizing for probabilities within each parse of each constituent regarding its pre-terminal, lexical head, and further lower level constituents (if any) probabilities.