COM6115: Lab Class 4

Zipf's Law

This week's lab introduces you to a well-known *empirical law* that is claimed to hold for language, called **Zipf's law**. The law implies a highly skewed distribution over linguistic elements (in this case words, but we can also apply it to other elements - see "Further Work", below). This skewed distribution is significant in many contexts (e.g. in regard to the effectiveness of word-oriented text compression methods).

We will attempt to verify the correctness of this law through some simple data analysis and graph plotting. Along the way, we will gain some familiarity with the graph plotting facilities available in Python.

Zipf's law is an empirical law formulated by the American linguist George Kingsley Zipf (1935). It states that in a large corpus, the **frequency** of any word is inversely proportional to its **rank** in the frequency table. Thus the most frequent word will occur approximately twice as often as the second most frequent word, three times as often as the third most frequent word, etc.

In the Brown Corpus, for example, the most common word (*the*) accounts for nearly 7% of all word occurrences, the second (*of*) for 3.5%, and so on. Consequently, only the 135 top-ranked vocabulary items are needed to account for *half* the word occurrences in the Brown Corpus.

Zipf's law is an instance of a **power law**. Similar power law observations have been made across many different sorts of data, unrelated to language. See the Wikipedia entry for Zipf's law to read more about this or, for an entertaining account, watch "The Zipf Mystery" on Vsauce: https://youtu.be/fCn8zs912OE.

Exercises

Simple graph plotting using Pylab

Pylab is a library of existing Python code, known as a **module**, which provides lots of useful functionality, including graph plotting.

The basic plotting function requires two arguments: a list of the x-coordinates of the points to be plotted, and a list of the corresponding y-coordinates (which should be the same length as the first list, of course).

To plot a graph with (x, y) points (0, 1.2), (1, 2.2) and (2, 1.8), for example, we form a list of the x values `[0,1,2]` and y-values `[1.2, 2.2, 1.8]`.

We could then plot this graph with the following code:

```
import pylab as p
X = [0, 1, 2]
Y = [1.2, 2.2, 1.8]
p.plot(X,Y)
p.show()
```

The `plot` command takes the lists of x and y coordinate values as stated above. The show command causes the graph to be actually drawn and displayed.

1. Try this code out yourself. If you're using the IPython Console in Spyder, graphs will display in the Console window. If you're running code in a terminal, however, graphs will display in separate windows (and you may need to click on the relevant icon at the bottom of your screen to bring them to the front).

Get the data

As a basis for evaluating Zipf's law, we need some word frequency data from a reasonably sized body of text.

2. Download the file `mobydick.txt` from the lab class folder in Blackboard.
3. As you can see by opening the file, it's the text of *Moby Dick* by Herman Melville (with some additional information and quotes about whales at the beginning).

Count and plot word frequencies

4. Write a Python script to count all the word token occurrences in the file.
   - Take a simple view of *tokenization* — simply use a regex to extract the maximal alphabetic sequences and treat these as the words, as we did in Lab 2.
   - Map the input to *lowercase*, so that we will treat e.g. 'The' and 'the' as the same word.
   - Count the words into a dictionary, and then produce a list of the words sorted in *descending order* of frequency count.
   - Get your script to print out:
     - The total number of word occurrences from the data file
     - The number of distinct words found
     - The top 20 words with their frequencies

   Unsurprisingly, the top 20 words are typical **stop words**. For this task, however, it makes sense to retain these words.

4. For what follows, we can forget the actual words from the data, and just use their frequencies, sorted into descending order. Begin by plotting these sorted frequencies against their rank position, i.e. where the most common word has rank 1, and so on. Try plotting this graph for different numbers of words, i.e. for the top 100, 1000, or the full set.
5. Another thing to plot is **cumulative** count, i.e. for rank 1 plot just the original frequency, for rank 2, the sum of ranks 1 and 2, and so on. This plot allows us to see how rapidly the occurrences accounted for by the top N ranks approach the total occurrences in the data.
6. As discussed on the Wikipedia page for Zipf's law, **power law** relationships are most easily observed by plotting the data on a log-log graph, with the axes being log(rank order) and log(frequency). If the data conform to Zipf's law, we expect this plot to be approximately linear (a straight line). Plot a graph of this relationship for the Moby Dick words.

Further Work

Adjusting plot appearance

By default, Pyplot draws graphs with a continuous line, in a random colour. This is fine for the Zipf plotting task, but in other cases you may want other formats. For this, `plot` takes an optional third argument, which is a string which allows us to control the plot format.

For example, in `plot(X,Y,'ro-')`, the format string `'ro-'` gives a red (`'r'`) continuous line (`'-'`) with circles (`'o'`) marking the data points, but we could instead choose blue (`'b'`) or green (`'g'`), asterisks (`'*'`) or crosses (`'x'`), or a line that is dashed (`'--'`), dot-dashed (`'-.'`) or absent (i.e. only showing data points).

Other functions (`xlabel,ylabel`) allow us to assign labels to the x and y axes (e.g. `xlabel('time')`), give the figure a title (`title`), or name a file in which the figure is saved (`savefig`) for later use, as a PNG image file.

We can plot more than one line on a graph by having more than one call to the `plot` function before calling `show`. A call to `figure` between `plot` calls causes a new figure to be started, so that multiple figures are displayed when `show` is called. We can use the `subplot` function to arrange multiple graphs within the same figure. For example:

```
import pylab as p
X = [0, 1, 2]
Y1 = [1.2, 2.2, 1.8]
Y2 = [1.5, 2.0, 2.6]
p.plot(X,Y1)
p.figure()
p.plot(X,Y2)
p.show()
```

```
import pylab as p
X = [0, 1, 2]
Y1 = [1.2, 2.2, 1.8]
Y2 = [1.5, 2.0, 2.6]
p.subplot(211)
p.plot(X,Y1)
p.subplot(212)
p.plot(X,Y2)
p.show()
```

See e.g. https://matplotlib.org/stable/tutorials/pyplot.html and other online tutorials to learn about additional plotting functionality.

Counting bigrams with NLTK

NLTK frequency distributions

In this lab, you were asked to count the number of times each token in a text corpus appeared, using a Python dict to store the words and their counts. This is a very common task in text processing, and so NLTK has a built-in class to perform this task. It is called `FreqDist`, for 'frequency distribution', and it takes a list of items (which might be words, tokens or some other unit) as input. Using this method, we can count the words in the webtext corpus easily:

```
import nltk
from nltk.corpus import webtext

words = webtext.words()
fd = nltk.FreqDist(words)

#test for occurrences of the word dog
print fd['dog']
```

This should output a number (120) – the number of occurrences of the word 'dog' in the NLTK webtext corpus.

The object stored as `fd` works in some ways like a standard Python `dict()` but also has a few extra options, e.g. a method `plot()` to print graphs of the frequencies computed.

The following code, for example, plots of graph of the 20 most frequent items counted, in descending order of frequency:

```
fd.plot(20,show=True)
```

The output looks like this:

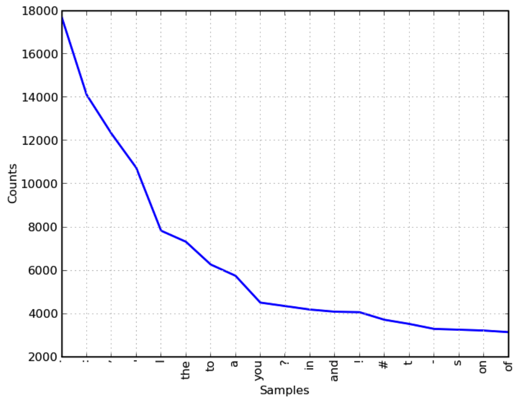

Figure 1: Top 20 word frequencies in the NLTK webtext corpus

The most common tokens, according to this graph, are punctuation. To restrict attention to just words, we might reduce the word list to just the alphabetic elements, using a list comprehension as follows prior to the counting step:

```
words = [w for w in words if w.isalpha()]
```

If we preferred to count tokens irrespective of case, we might extend this to map tokens to be lower-case, as follows:

```
words = [w.lower() for w in words if w.isalpha()]
```

(Note: list comprehensions compute and return the list described. In many situations, we do not actually need this list to be constructed, as we only want to iterate over the elements that appear within it. Python provides a related notation for creating **generators** for exactly this purpose. The details are beyond the scope of this lab class, but see e.g. this page.)

Counting N-grams

NLTK provides several helpful methods for producing bigrams, trigrams, and N-grams of any size you specify, which are called (appropriately) `nltk.bigrams()`, `nltk.trigrams()` and `nltk.ngrams()`. Each takes a list of words as argument, whilst the `ngrams()` function takes an additional argument for N-gram length. These methods quickly produce large lists that take up lots of memory and that are slow to process. To avoid this problem, you can instead use their **iterator** variants, e.g. `nltk.ibigrams()`.

We can graph the top bigrams in a section of the Brown Corpus using a similar approach to that used above. The Brown Corpus is a large corpus containing a wide variety of text. Try typing `nltk.corpus.brown.categories()` for a list. You may specify one “category” of the Brown corpus to use, or a list of multiple categories. If no categories are specified (i.e. `brown.words()`) then the entire corpus will be used.

```
from nltk.corpus import brown
fd_bigram=nltk.FreqDist(
    nltk.bigrams(brown.words(categories='editorial'))
)
```

What if we want to exclude all bigrams that contain punctuation? Or bigrams that contain very common words? NLTK contains a corpus of very common words (or “stop words”) that we can use to make this easier.

```
stops = nltk.corpus.stopwords.words('english')
stops += list('"\',./;:-)(?!') # single char punctuation
stops += ['``',"''",'--']      # multi-char punc tokens
stops = set(stops)

bigrams = \
    [(w1.lower(),w2.lower())
     for (w1,w2) in nltk.ibigrams(brown.words(categories='editorial'))
     if not (w1.lower() in stops or w2.lower() in stops)]

fd=nltk.FreqDist(bigrams)

print fd.items()[:10]
```

Extra exercises

7. Write a program to find all trigrams that occur at least three times in the Brown Corpus.
8. Use NLTK to examine frequencies of bigrams and trigrams in the Moby Dick data. Does Zipf's law hold for these n-grams?