COM6115: Lab Class

Zipf's Law

1. This week's lab introduces you to a well-known *empirical law* that is claimed to hold for language, known as **Zipf's law**, which describes a *highly-skewed distribution* over linguistic elements (e.g. *words*) that is significant in many contexts (e.g. the effectiveness of word-oriented text compression). We will seek to verify this law, through simple data analysis and graph plotting. Along the way, we will gain some familiarity with the graph plotting facilities available within Python.

2. Zipf's law is an empirical law formulated by the American linguist George Kingsley Zipf. 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 around 6% of all word occurrences, the second (*of*) for 3%, and so on. Consequently, only the 135 top-ranked vocabulary items are needed to account for *half* of all word occurrences in the corpus.

   Zipf's law is an instance of a *power law*. Similar power law observations arise for other sorts of data, unrelated to language. See the Wikipedia entry for Zipf's law for more about this.

3. As a basis for evaluating Zipf's law, we need access to a reasonably sized body of text, from which to compute word counts. Download the code/data file (zip) for this lab from Blackboard, and unzip it. The file `Zipf_lab_STARTER_CODE.py` contains an initial fragment of code, which downloads the text of the Brown Corpus (one of the first major NLP corpora, with ∼1.1M tokens), and sets the variable `brown_words` to be a list (or strictly, an *iterator*), over which you can iterate (with a `for`-loop) to get all the tokens of the Brown Corpus text, one at a time.

4. Extend the *starter code* to loop over this data, mapping tokens to *lowercase* (to conflate case variants), and counting them into a dictionary. Your script should print the total count of all tokens in the data, and of *distinct* tokens found, plus the 20 most common tokens with their counts. You will see this list includes *punctuation* items. You might (optionally) modify your code to skip over these items during counting (e.g. using a regex). Unsurprisingly, the words in this top 20 are typical *stop-words*. For this task, however, it makes sense to *retain* these words.

5. For what follows, we can forget the actual words of the data, and just use their counts, sorted into *descending* order. (For dictionary `d`, calling `d.values()` gives the *values* stored in `d`.) Firstly, plot these sorted frequencies against their rank position, i.e. so the most common word has rank 1, and so on. Some notes of simple graph plotting in Python are provided on the next page. Try plotting this graph for different numbers of words, i.e. for the top 100, 1000, or the full set.

6. 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.

7. As the Wikipedia page for Zipf's law indicates, *power law* relationships are best observed by plotting the data on a log-log graph, i.e. with axes log(rank order) vs. log(frequency). The data conform to Zipf's law to the extent that the plot is linear. Plot a graph of this relationship.

8. Next modify your script to count all the word *bigrams* in the Brown Corpus, and re-plot the above graphs for the bigram counts, to see if they also obey Zipf's law. (A simple way to count bigrams is to keep a copy of the word seen in the last iteration of the counting loop, e.g. as variable `last_wd`. When the next word (`wd`) is read, you can take the tuple "`(last_wd,wd)`" to be the bigram to count, using it as the key in the counting dictionary.)

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-coords (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 coord values as stated above. The `show` command causes the graph to be actually drawn and displayed. 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).

By default, lines are drawn 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(Xs,Ys,'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/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. `http://matplotlib.org/users/pyplot_tutorial.html`, and other online tutorials to learn about additional plotting functionality.)