

COM3110/4115/6115: 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**. The law implies a *highly-skewed distribution* over linguistic elements (in this case *words*) that 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 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 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.
3. As a basis for evaluating Zipf's law, we need some word frequency data from a reasonably sized body of text. For this purpose, download the file `mobydick.txt` from the module homepage.
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. Also map the input to *lowercase*, to conflate case variants. 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, and also 20 words with their frequencies. Unsurprisingly, these 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 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. 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 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). The data conform to Zipf's law to the extent that the plot is linear. Plot a graph of this relationship.

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.)