LING 227/627 — Language and Computation I
Spring 2021

Problem Set 1 (Part B): Zipf and Eliza
Due 2/18/21 at midnight

1 Zipf's Law

In class, we discussed Zipf's law, according to which the frequency of a word $f(w_i)$ is roughly inversely proportional to the rank of word (where rank is the position of the word in the ordering of words by frequency).

$$f(w_i) \approx \frac{k}{\mathsf{rank}(w_i)}$$

In this problem, you will explore this relationship, by writing a Python program that produces a log-log plot of word rank by frequency for a number of corpora, and see if they give (roughly) linear relationships. You will find it convenient to work in Jupyter, and you should submit a Jupyter notebook as your solution for this problem.

1. Choose an English corpus from among those provided with NLTK. Load the corpus into python and use NLTK tools to construct a frequency distribution over the words in this corpus. Though you can use the NLTK plotting function we discussed in class to produce a plot of this distribution, you will need to use a different tool to make a log-log plot. I recommend the `matplotlib` library, which you can load as follows:

```
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt
```

   The first line tells `matplotlib` to generate plots directly in the Jupyter notebook rather than in a pop-up window. The `matplotlib` library provides a range of plotting functions, but the one you will find most helpful is `plt.loglog`, which takes two arguments that are each lists of numbers, representing the values on the x- and y-axes, respectively, for the data points to be plotted.[1] In order to use this plotting function, you will need to extract the frequency values from the `FreqDist` you construct. In addition, you will need to construct a list of ranks to give as the first argument to the plotting function.

   To produce a more readable plot, you may find it helpful to label the axes and include grid lines. So, you might use commands like the following to produce your plot.

[1] This library also provides a more basic plotting function `plt.plot` which will produce a plot similar to the NLTK-provided plotting function.

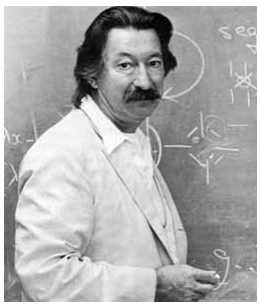
```
plt.loglog(ranks,freqs)
plt.ylabel('frequency(f)',fontsize=14,fontweight='bold')
plt.xlabel('rank(r)',fontsize=14,fontweight='bold')
plt.grid(True)
```

2. You will now see if the same pattern holds in data from other languages. To do this, download one of the parallel corpora of European Parliamentary proceedings, which are available at `http://www.statmt.org/europarl/`. You should choose a pair of languages (English and one other), and plot your log-log word rank by word frequency plot for both English and Non-English texts. Usefully, these are parallel corpora, so any differences between them are not the result of differences in the properties of the texts. Do you find the same Zipfian pattern in each one? Discuss any differences you find, and possible reasons for these differences. You can put these comments directly into the Jupyter notebook, as a markdown field.

3. Finally, using one of the English corpora you have already explored (either from NLTK or Europarl), construct a frequency distribution of alphanumeric characters (ignore punctuation symbols), and once again construct a log-log for this frequency distribution. Do you find the familiar Zipfian pattern here?

2 Modifying the "Eliza" program

On the first days of class, I briefly mentioned Joseph Weizenbaum's Eliza program that mimics a Rogerian psychotherapist through simple textual manipulation. In this problem, you will modify an implementation of a simple Eliza program that I have provided for you in the file `eliza.py`. You can run the Eliza program by typing `python eliza.py` at the command line. Type `quit` to leave the program.



```
$ python eliza.py
Hello! I am the psychotherapist. Please describe your problems to me.
> I need to figure out how to process language
Why do you need to figure out how to process language?

> Because it sounds like fun
Is that the real reason?
```

```
> quit
Thank you. That will be $150. Have a nice day!
```

Note that because we're keeping things simple for now, things work best if you don't put any punctuation at the end of your inputs.

Open the file `eliza.py` in your editor to have a look at the code. The file includes four functions as well as a loop that handles the interaction with the user. The function `give_response` is the core of the Eliza program: given a string as input, it constructs a string as response on the basis of a variety of conditions on the input string. This function uses a number of auxiliary functions: `after` takes a string and a prefix, and returns the remainder of the string that follows the prefix, `reflect` applies the `reflect_word` function to each of the words in a string. This latter function modifies pronouns and verbs so that they make sense as responses from Eliza.

1. Check what Eliza's response is if the user says "I can't stop myself from talking to psychotherapists," and modify the `reflect_word` function to correct the problem you discover.

2. Modify the `response` function so that whenever the user says something of the form "I never ...", Eliza responds according to the following pattern:

```
> I never eat cake
You really never eat cake at all?

> I never go snorkeling on Tuesdays
You really never go snorkeling on Tuesdays at all?
```

3. One thing that seems very unnatural is that Eliza gives exactly the same response ("Could you elaborate on that?") to anything that doesn't fit one of the recognized patterns. Of course one way to improve things is to specify a longer list of recognized patterns — but a sneaky simpler alternative is to select in some random-seeming way from a larger range of completely general responses ("How do you feel when you say that?", "Perhaps the answer lies within yourself?", etc.). Modify the `response` function so that Eliza says "Could you elaborate on that?" if the length of the user's input is 0, 3, 6, 9, etc.; says something else if the length of the user's input is 1, 4, 7, 10, etc.; and says something else again if it is 2, 5, 8, 11, etc. You can use the Python mod operator `%` for this: given two integers `a` and `b`, `a % b` returns the remainder when the first is divided by the second.

4. It would be nice if Eliza could give responses like the following any time the user's input contains the word "is":

```
> my mother is a hamster
```

```
Why is your mother a hamster?

> my brother is going snorkeling next Tuesday
Why is your brother going snorkeling next Tuesday?
```

Your friend, Fred, has a clever idea about this: we can produce these responses by starting with "Why is ", then repeating the part of the user's input that comes before the first occurrence of the word "is", and then the part of the user's input that comes after it. Fred's linguist friends try to tell him that this is a bad idea, but he's very stubborn and doesn't listen.

To implement Fred's idea, first write a function `before` which can be used to pick out the part of a string which precedes the first occurrence of a certain target, analogous to `after`. (Hint: look at what the `split` method used in `after` returns.) Don't worry about what happens if someone calls your `before` function with two strings where the second one doesn't appear anywhere in the first one.

Now modify the `response` function to produce the kind of question-answer pairs Fred has in mind.

```
> my mother is a hamster
Why is your mother a hamster?

> this issue is not resolved
Why is this issue not resolved?

> this issue concerns hamsters
Could you elaborate on that?

> the goldfish that my mother's sister sold to John is drowning
Why is the goldfish that your mother's sister sold to John drowning?
```

What are some inputs for which Fred's rule will produce ungrammatical responses? Don't worry about minor issues like capitalization or punctuation — we're looking for cases where the rule produces "word salad" output.

That's it!

Please upload to the Canvas assignment your Jupyter notebook for the Zipf problem, called `zipf.ipynb` and your revised `eliza.py` file for the Eliza problem. Your programs should be commented as appropriate to explain anything unusual in your code and to answer questions raised the problem set.