Homework #1: CMPT-413

Anoop Sarkar
http://www.cs.sfu.ca/~anoop

Only submit answers for questions marked with †. For the questions marked with a ††; choose one of them and submit its answer. Typically, $\dagger\dagger^e$ is easier and $\dagger\dagger^h$ is harder.
Copy the files for this homework: `scp -r fraser.sfu.ca:/home/anoop/cmpt413/hw1 .`
Put your solution programs in the `hw1/answer` directory. There are strict filename requirements. Read the file `readme.txt` in the `hw1` directory for details.
The `hw1/testcases` directory contains useful test cases; you will need to consult `readme.txt` for the mapping between the homework questions and test cases.

Important! To solve some questions in this homework you must read the following chapters of the NLTK Book, available online at *http://www.nltk.org/book*

- Chapter 1. Language Processing and Python
- Chapter 2. Accessing Text Corpora and Lexical Resources

(1) Define a variable silly to contain the string: 'newly formed bland ideas are unexpressible in an infuriating way'. Now write code to perform the following tasks:

   a. Split silly into a list of strings, one per word, using Python's split() operation.

   b. Extract the second letter of each word in silly and join them into a string, to get 'eoldrnnnna'.

   c. Build a list phrase4 consisting of all the words up to (but not including) 'an' in silly. Hint: use the index() function in combination with list slicing.

   d. Combine the words in phrase4 back into a single string, using join(). Make sure the words in the resulting string are separated with whitespace.

   e. Print the words of silly in alphabetical order, one per line.

(2) Write code to abbreviate English text by removing all the vowels `'aeiou'`.

(3) Write code to read a file and print it in reverse, so that the last line is listed first.

(4) Rewrite the following nested loop as a nested list comprehension:

```
words = ['attribution', 'confabulation', 'elocution',
    'sequoia', 'tenacious', 'unidirectional']
vsequences = set()
for word in words:
    vowels = []
    for char in word:
        if char in 'aeiou':
            vowels.append(char)
    vsequences.add(''.join(vowels))
print sorted(vsequences)
# prints:
# ['aiuio', 'eaiou', 'eouio', 'euoia', 'oauaio', 'uiieioa']
```

(5) † Write a Python program to eliminate duplicate lines from the input file. This process is called *deduplication* which is particularly useful in very large text data-sets.

(6) † Read the Wikipedia entry on the Soundex Algorithm. Implement the algorithm in Python which accepts a string from standard input and prints out the Soundex value. See the `testcases` directory for examples.

(7) Download the front page from *http://www.cbc.ca* and print out the contents to standard output (sys.stdout) and print out to standard error (sys.stderr) the number of characters, words and lines on the page. You can optionally strip out the html tags. Here's what the output should look like (number of chars, number of words, number of lines):

```
$ python urlwc.py | wc -l
17145 1114 1196
    1196
```

(8) Use the function `fisher_yates_shuffle` given below in a Python program which reads in a file and prints out each line with the words randomly shuffled. Each line of text should appear in the same order as the input but the words in each line should be randomly permuted.

```
from random import *

# generator for a reverse iterator over an array
def reverse_iterator(data):
  for index in range(len(data)-1, -1, -1):
    yield index

# fisher yates random shuffle of an array
def fisher_yates_shuffle(array):
  for i in reverse_iterator(array):
    j = int(randrange(i+1))
    (array[i],array[j]) = (array[j],array[i])

if __name__ == "__main__":
  x = range(1,51)
  fisher_yates_shuffle(x)
  print " ".join(str(i) for i in x)
```

(9) † Using `nltk.corpus.gutenberg.words('austen-sense.txt')`, collect the frequency of each word in the 'austen-sense.txt' corpus (Sense and Sensibility by Jane Austen) and print it out sorted by descending frequency. Print out the rank of the word, the frequency and the word itself. The first few lines of the output should look like this:

```
$ python2.6 freq.py | head
1 9397  ,
2 4063  to
3 3975  .
4 3861  the
```

We say that ',' has rank of 1 and 'to' has rank of 2, and so on.

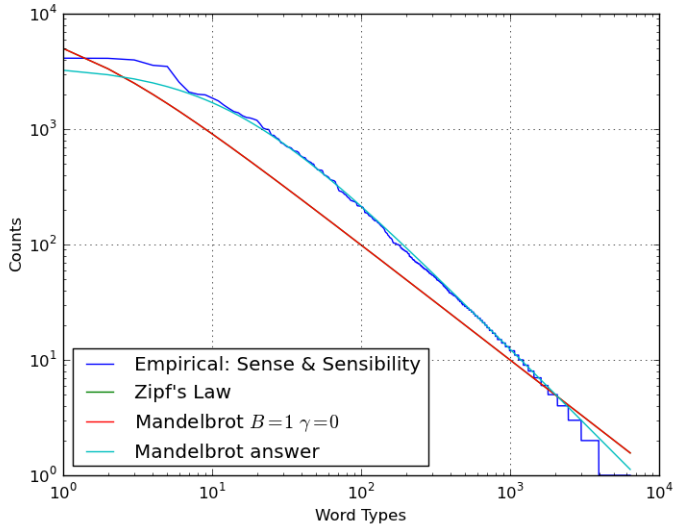

Figure 1: Log-log plot of word rank (x-axis) against frequency (y-axis), Zipf's formula and Mandelbrot's formula (using some arbitrary values for P, B and γ).

(10) **Zipf's Law**: The rank r of each word (see Q 9) is based on its frequency f. Zipf's law states that $f \propto \frac{1}{r}$ or, equivalently, that $f = \frac{k}{r}$ for some constant factor k. For example, if Zipf's law holds then the 50th most common word should occur with three times the frequency of the 150th most common word. This relationship between frequency and rank was first noticed by J. B. Estoup in 1916, but was widely publicized by Zipf in his 1949 book: *Human Behaviour and the Principle of Least Effort*.

Now compare Zipf's formula with the Mandelbrot formula $f = P(r+\gamma)^{-B}$ which can be written as:

$$log(f) = log(P) - B \cdot log(r+\gamma)$$

Use the Python code in `freqdist_plot_example.py` as a template: Save your graph as a PNG file using the `pylab.savefig` function.

For the following questions, use the text of Sense and Sensibility and collect the empirical word frequencies from the text using the solution to Q (9).

a. Change the parameters P, B and γ in the Mandelbrot formula to match the empirical distribution of word frequencies.

† Submit the Python code that plots the empirical rank vs. frequency plot, the Zipf's formula plot and the Mandelbrot's formula plot which has been modified to match the empirical distribution. Submit the Python code and the PNG file of the plot. It should look like Fig. 1.

b. What happens to the Mandelbrot formula when $B = 1$ and $\gamma = 0$?

c. † A stemmer is a program that uses heuristic rules to convert a word into a stem (the word minus any characters that belong to the prefix or suffix). Use the Porter stemmer to plot the empirical distribution using stems instead of words. Submit the Python code that creates the plot and the PNG file of the plot.

The following code prints the stem for the word *walking*:

```
from nltk import stem
p = stem.PorterStemmer()
print p.stem("walking")
```

(11) The following Python program prints out a dispersion plot: visually depicting how often and when a certain character occurs in the Jane Austen novel 'Sense and Sensibility'.

```
from nltk.corpus import gutenberg
from nltk.draw import dispersion
words = ['Elinor', 'Marianne', 'Edward', 'Willoughby']
dispersion.dispersion_plot(gutenberg.words('austen-sense.txt'), words)
```

Compare the dispersion plot for the words *walking, talking, hunting* compared to all of the various inflected forms of the stems *walk, talk, hunt* that actually occur in the novel. Use the Porter stemmer to map from stems to inflected forms.

(12) † Write a Python program to report the line numbers where each pair of adjacent words occurred in a text. The user can set a minimum frequency for each word in the pair using a `-n` parameter, so the only word pairs that are reported are those in which each word in the pair occur with frequency ≥ the minimum frequency.

(13) †††^{e} **Sanskrit Prosody**

Pingala, a linguist who lived circa 5th century B.C., wrote a treatise on prosody in the Sanskrit language called the Chandas Shastra. Virahanka extended this work around the 6th century A.D., providing the number of ways of combining short and long syllables to create a meter of length n. Meter is the recurrence of a pattern of short and long syllables (or stressed and unstressed syllables when applied to English). He found, for example, that there are five ways to construct a meter of length 4: V_4 = {LL, SSL, SLS, LSS, SSSS}. In general, we can split V_n into two subsets, those starting with L: {LL, LSS}, and those starting with S: {SSL, SLS, SSSS}. This provides a recursive definition for computing the number of meters of length n. Virahanka wrote down the following recursive definition on his trusty 6th century Python interpreter:

```
def virahanka(n, lookup = {0:[""], 1:["S"]} ):
    if (not lookup.has_key(n)):
        s = ["S" + prosody for prosody in virahanka(n-1)]
        l = ["L" + prosody for prosody in virahanka(n-2)]
        lookup.setdefault(n, s + l) # insert a new computed value as default
    return lookup[n]
```

Note that Virahanka has used top-down dynamic programming to ensure that if a user calls `virahanka(4)` then the two separate invocations of `virahanka(2)` will only be computed once: the first time `virahanka(2)` is computed the value will be stored in the `lookup` table and used for future invocations of `virahanka(2)`.

Write a *non-recursive* bottom-up dynamic programming implementation of the `virahanka` function. You *cannot* use any memo-ization. The output should look like this:

```
virahanka(4) = ['SSSS', 'SSL', 'SLS', 'LSS', 'LL']
```

(14) †††^{h} **Chinese Word Segmentation**

In many languages, including Chinese, the written form does not typically contain any marks for identifying words (compared to the space character which delimits words in the English script). Given the Chinese text "北京大学生比赛", a plausible segmentation would be "北京(Beijing)/大学生(university students)/比赛(competition)" (Competition among university students in Beijing). However, if "北京大学" is taken to mean Beijing University, the segmentation for the above character sequence might become "北京大学(Beijing University)/生(give birth to)/比赛(competition)" (Beijing University gives birth to competition), which is less plausible.

Word segmentation refers to the process of finding blocks that correspond to words in a character sequence. Word segmentation is useful in many natural language processing applications such as machine translation.

Your program should use the data `zh-wseg.train.utf8` which contains (some) Chinese words and their frequencies. You can solve this task in any way you wish. You cannot use any other data source apart from the one given to you, but you can add in specific rules for classes of characters like numbers.

For checking your output, we do *not* do an exact match with the true segmentation (which was produced by a human expert) since even small errors result in a segmentation that is incorrect. Instead we match the words produced by the system and check for overlap with the words in the true segmentation (which we call by different names: gold data or reference data or the "truth"). The score reported by the check program is the *F-measure* which is computed in the check program using the `f_measure` function defined in `http://nltk.org/api/nltk.metrics.html`.

You might want to take inspiration from the following book chapter which talks about word segmentation algorithms (for English):

> Chapter on *Natural Language Corpus Data* by Peter Norvig in *Beautiful Data* (Segaran and Hammerbacher, 2009) http://norvig.com/ngrams/