Homework #1: CMPT-413

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

Only submit answers for questions marked with †.
Read the instructions on how to use svn. In your svn repository, 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 (optional if you already know some Python)
- Chapter 2. Accessing Text Corpora and Lexical Resources
- Chapter 3. Processing Raw Text

(1) Write code to abbreviate English text by removing all the vowels `'aeiou'`. Only lowercase vowels should be removed.

(2) 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. The original order of the lines should be retained as this is often useful information that might be useful.

(3) 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. The most frequent word has rank of 1, the second most frequent has rank of 2, and so on. Print out the rank of the word, the frequency and the word itself. It is not essential but you should try to use `nltk.probability.FreqDist`.

(4) 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)
```

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). A popular, but not very accurate, stemmer used widely is the Porter stemmer. The following code prints the stem for the word *walking*:

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

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.

(5) † **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):
    if n == 0: return [""]
    if n == 1: return ["S"]
    s = ["S" + prosody for prosody in virahanka(n-1)]
    l = ["L" + prosody for prosody in virahanka(n-2)]
    return s+l
```

Note that Virahanka has used recursion which leads to some duplication of effort. Notice that for `virahanka(4)` there will be two separate invocations of `virahanka(2)`. This can be avoided by storing the value of `virahanka(2)` in a table and simply looking up the value in this table when it is needed multiple times.

Write a *non-recursive* bottom-up dynamic programming implementation of the `virahanka` function. You *cannot* use any memo-ization.

Start with the `virahanka_stub.py` and modify the `virahanka` function. Do not delete the recursion check. If you do we cannot auto check your submission.

(6) † **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.

The input will be a sequence of Chinese characters without spaces between words, and the output should contain ASCII spaces between the words found by your segmentation program. Looking at the testcases will make this clear.

Your program should use the data `zh-wseg.train.utf8` which contains (some) Chinese words and their frequencies. 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 check correctness by performing an exact match with the true segmentation since even small errors result in a segmentation that is incorrect. Instead we check for overlap with the words in the true segmentation (which we refer to in different ways: as gold data or reference data or the "truth"). The score reported by the check program is *F-measure* that is computed using the `f_measure` function defined in `http://nltk.org/api/nltk.metrics.html`.

For the most challenging input file `zhwseg.in`, an F-measure of 80% or higher should be your goal. Your submission will be eventually graded on a hidden test set, but higher performance on the provided data

should also give you better performance on the hidden test set. To get you started, a baseline segmenter has been provided to you called `segment_stub.py` that gets about 28% F-measure on `zhwseg.in`.

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/