Software Design: Homework 5

Sarah Walters

March 3, 2014

1 Project Overview

This software enables a user to generate "lorem ipsum poetry" - semantically meaningless text that fits a specified stress and rhyme scheme - using words sourced from a novel.

I developed the code in stages. First, I built a module which describes how words sound: initially, it imported the phoneme set cmudict from the nltk corpus, split words into syllables according to the structure of English pronunciation, then detected exact rhyme using simple string matching. I then modified the rhyme detector such that it accounts for slant rhyme using a memoized Levenshtein distance calculator and added a method which extracts the stress pattern from a word - that is, which syllables are stressed and which are unstressed when the word is pronounced. After doing so, I built a module which combs through a .txt file downloaded from Project Gutenberg. a collection of freely available ebooks, and produces a dictionary which describes the number of times each word in the file appears. Lastly, I integrated the functionalities of the two modules into a third module which takes a structure of stress patterns and rhyme scheme as input and produces a randomized poem using the words in the specified .txt file.

2 Implementation

2.1 Structure

The code I wrote consists of three modules. The first, gutenberg, includes functionalities related to reading from a Project Gutenberg .txt file. The second, rhyme, produces analysis related to how words sound - rhymes, stresses, etc - using the phoneme set cmudict from the nltk corpus. The third, generate, implements functionalities defined within rhyme and gutenberg in order to produce a poem which matches a pre-defined rhyme and stress scheme.

2.2 Poem generation

The process of generating a poem involves first constructing two dictionaries related to the words in a text (obtained from a .txt file using the module gutenberg.) The first dictionary, wts, maps from word to stress pattern. It relies on the function stresses contained within the module rhyme, which reduces a word into a string of 1's and 0's representing stressed and unstressed syllables (eg stresses('dictionary') produces '1010'.) The function stresses draws from the set of phonemes defined by cmudict: all vowel phoneme strings contain a number (either a 0, representing unstress, a 1, representing primary stress, or a 2, representing secondary stress), so it is possible to parse stress pattern once the syllable-related functions contained within the module rhyme split a word into its syllables. The second dictionary, stw, then combs through all of the possible stress patterns for varying pronunciations of each word ('history', for example, can be pronounced either ['HHIH1S', 'TER0', 'IY0'] or ['HHIH1S', 'TRIY0'], resulting in stress patterns of '100' and '10' respectively), and creates a map from stress pattern to words which match that pattern.

Then, it is necessary to generate the appropriate number of appropriately-sized sets of rhyming words. rhymeSets, a generic function which does this, takes an input matrix N whose length definese the number of sets which are generated and whose elements define the sizes of those sets, then relies on a global dictionary, rhymes, which maps from a representative word to a number of rhyming words, establishing unique sets. endRhymeSets, a specific function which calls rhymeSets, uses a parameter string consisting of 1's and 0's

representing the syllable pattern at the end of all of the lines in the poem in order to build the global dictionary rhymes such that all of the sets of rhyming words have stress patterns which fit the end of the syllable pattern.

The function poem performs the actual generation: it takes as a parameter a list of lists where each sublist represents a line in the poem and contains both a string of 1's and 0's (eg '01001001') representing the line's stress pattern and a rhyming character (eg 'A') which defines which other lines in the poem the line should rhyme with. It builds a rhyme scheme by grouping the rhyming characters, then fills in the rest of each line by removing substrings from the stress pattern as it adds to the line using the dictionary stw (which maps from stress to words). Lastly, the module includes a functionality which choose two words from the poem to use as a title.

To generate a poem, open the file generate.py, scroll all the way down, set the variable filename to the name of the .txt file from which you wish to read, then choose a poem format. Run the script generate from the commandline.

3 Results

The code produces a semantically meaningless poem which matches a particular rhyme scheme and meter and which is constructed from the words in a text file. I've pre-defined functions which produce sonnets and limericks, and it would be trivial to add a wider range of options. Here is an instance of a sonnet (each line has an iambic pentameter stress pattern, or '0101010101') generated from Charles Dickens' "A Tale of Two Cities", with rhyme scheme called out by rhyming characters in parentheses after each line:

Speculate Deliberating

Extraordinary swung and soothed release (A)
Regarding overcoming would her last (B)
Exaggerated sadness blast police (A)
Relationship obtain conveyance hast (B)
Congratulations disinclined i'm crushed (C)
Are reservoirs corroborate forgive (D)
Attenuated correspondence brushed (C)
Aristocrats de laws attended give (D)
Exterminating tavern heavy sloth (E)
For bared were intermission justice spring (F)
Oppression revolutionary oath (E)
Identify deliberating ring (F)
Examination comprehend years sneezed (G)
Disturbed obeying speculate displeased (G)

–Robot Frost, 'Poems Ipsum' 2014

Here is an instance of a limerick (long lines with stress pattern '01001001' and short with stress pattern '01001') generated from Dostoevsky's "Crime and Punishment," once again with rhyme scheme called out:

Unwelcome Blames

Complain incomplete because names (A)
Identification for blames (A)
Philosopher groaned (B)
s Has glittering moaned (B)
Unwelcome who've brag isn't flames (A)

–Robot Frost, 'Poems Ipsum' 2014

4 Reflections

My project didn't have a well-defined scope to begin with - I iterated through several ideas to arrive at this one - but once I settled upon poem generation based on meter and rhyme scheme, the project was indeed appropriately scoped - thought-provoking and challenging, but doable. I chose to work alone on this assignment and in doing so opted out of the opportunity to practice coordinating on a software project with a partner, and I've had enough experience writing software that I've established a process which works well for me personally; for that reason, I think I would have learned more had I chosen not to work alone, and I am looking forward to doing so in the context of the next assignment.