Ryan Vasios
CS 150: Natural Language Processing
Final Project Report

Abstract: My final project dealt with the creation of an original piece of software that studied and generated poetry. I then evaluated the project by Turing testing it on friends. Much to my surprise, participants saw a lot more humanity in the generated pieces than I had anticipated.

Introduction: I'm very fortunate that this course offered me an opportunity to develop a piece of software I'd wanted to play with for a while, a machine that writes poetry. When I originally expressed my curiosity to a close friend, he expressed surprise and even disdain for the idea. Poetry is, after all, about coming closer to your identity as a human being. Relegating its creation to a machine would seem to rob it of its dignity.

However, beyond being a testament to human emotion and life, poetry is a craft and skill, that necessarily has empirical features, namely in its more aural qualities. I wanted to see if a machine could imitate poetry through the use of rhyming schema, and string words together by features that are contextual, syntactic, and phonetic.

The tools that I drew on for this project include the Natural Language Tool Kit, The Carnegie Melon Pronounciation Dictionary, and Thomas Palgrave's 1875 anthology of English verse, The Golden Treasury.

Method:

-Parsing:

Our poetry corpus was prepared for use by lowercasing all words and removing all punction marks at the end of lines, within lines, and within words

-Rhyming Schema:

Rhymes were collected from our anthology by creating an extensive list of every word that ends every line. We then iterate through this list searching for proximal rhymes. That is, for every word in this list, we look at the all of the neighboring four words. Those words that both 1) share the same last emphasized syllable and 2) share all the same phonemes after this last emphasized syllable, were accepted as rhymes. This method worked very well with few exceptions. This method was able to extract 3015 distinct pairs of rhymes from our original corpus.

- **Form:**

Our poetry model selects pairs of rhymes randomly. Having some spontaneity in results contribute greatly to the fun of using this software. It then aligns the rhymes as couplets (AABBCC…) or as alternates (ABABCDCD). These correspond to the methods write_poemcouplets() and write_poemalt() and require that poem lengths be either %2=0 or %4=0 respectively.

-Premise of line construction:

The end of every line of verse is predetermine with the predetermined rhymeword. We then work our way backwards generating the rest of the line until we reach the determined syllable length of the line (stored in the global variable 'linelength'). The probability that a given word at n-1 precedes words(n, n+1, n+2) is the product of two other probabilities, that is the probability that a word's part of speech precedes the part of speech in front of it, and the probability that a word is phonetically consistent with the phoneme set of those words in front of it.

- Part of speech tagging and training:

I was originally hope that our friend the Penn Treebank would be useful in assigning parts of speech to the Golden Treasury. However, it became apparent that the vocabulary of both these collections were very disparate and I wasn't willing to compromise the rich, colorful vocabulary of the Golden Treasury.

Fortunately, NLTK contains a method for part of speech tagging, pos_tag(). Admittedly, this method isn't always so reliable, but it's the best I could find and I'd love to try improving upon it in the future.

The Penn Treebank was still a useful tool for my purposes of training my poetry model on POS transitions. This seemed like a better decision because as I've mentioned, NLTK's pos_tag() method isn't infallible and I wouldn't want it's defects to contaminate the POS transition training. Furthermore, poetry (especially if it's from such a varied period of time from 1600-1875) usually features more liberal syntax. I thought training it on the Penn Treebank would bring it a little 'closer to earth'.

I simply tweaked the MLE for bigrams so that the the

P(POSn-1 | POSn) = C(POSn-1, POSn) / C(POSn-1)

-**Phonetic likelihood:**

The Carnegie Melon Pronounciation Dictionary translates all words in the English language into a list of 39 possible phonemes. In my model, we computed the phonetic likelihood of a word being added to a line as follows:
(number of phonemes shared with the line's set of phonemes) / (size of line's set of phonemes)

If a word being examined shares no phonemes with the line, it's smoothed likelihood is computed as:

.01 / (size of line's set of phonemes)

-Eligible words for adjoining:

Instead of plowing through our entire vocabulary every time we want to add a word to the front of our line, we test words that share some 'context' with the current

head of the line. That is, any word that co occurred in a line of verse in the Golden Treasury.

-Algorithm structure:

Essentially line construction works through a queue of line's being developed as follows:

- Initially entry into the queue is a line consisting purely of the rhyme word
- We then examine every eligible 'context sharing' word, make a newline with this word preceding the current head of the line(rhyme in this case) and compute their probability as mentioned above:
- This list of possible (currently two word) lines is then sorted by probability. Only a preselected top-number of new lines are returned and added to the queue. This number of preselected lines is referred to as the 'branching factor' and stored in a global variable. Experience has shown that in generating longer lines, the branching factor must be curtailed (to say 2 or 3) to obtain feasible performance times .
- This process is repeated with the new lines added to our queue. We stop adding new lines if they are greater or equal to the preselected line syllable length, keeping those lines that are of the correct length.
- Once the queue is empty, we return the poetry line of the correct length with the greatest probability!

Experimental Setup:

At first look of my results, I was very impressed. I thought the resulting poems would be much more drivelly. One thing that stood out was the predilection for lines to prefer fewer, longer words to meet the syllable length since this would involve fewer probabilities. This lends a more humane quality to the lines generated, incorporating a more colorful vocabulary, novel tropes, and fewer conjunctions, articles, and pronouns, whose excessive presence could sound overly mechanical.

However, would people be willing to read what was generated and be willing to entertain it as actual poetry? That is, would people be willing to read the output and identify its author as human?

To test my model out. I created a Turing Test that I offered to 10 friends and relatives. It contains 5 poems created by my program and 6 excerpts of verse. I've attached a copy of it with the submission of this paper.

Participants were not told anything about the program, merely that they would have to distinguish verse engineered by humans from verse engineered by a computer. They were encouraged to spend no more than 2 or 3 minutes doing this as well. It's also worth pointing out that in order to confound participants, I used diverse styles of human

poetry. Number 3 (Ben Jonson) was nearly unanimously detected as human because its poetic form and grammatical quality is very traditional. However, other human samples were from writers who were stylistic experimenters, EE Cummings, Gertrude Stein, and Arthur Rimbaud (translated). In only case, did I break up a generated machine poem with a few extra new lines so as to diversify their appearances.

Results:

	1	2	3	4	5	6	7	8	9	10	11
ans	**C**	**H**	**H**	**H**	**C**	**C**	**H**	**H**	**C**	**H**	**C**
part1	**C**	**C**	**H**	**H**	**H**	**H**	**C**	**H**	**H**	**C**	**C**
part2	**C**	**H**	**C**	**H**	**C**	**H**	**C**	**C**	**H**	**C**	**C**
part3	**C**	**H**	**H**	**H**	**C**	**C**	**C**	**H**	**C**	**C**	**C**
part4	**C**	**C**	**H**	**H**	**H**	**C**	**H**	**C**	**C**	**C**	**C**
part5	**H**	**C**	**H**	**H**	**H**	**C**	**C**	**H**	**C**	**H**	**H**
part6	**C**	**C**	**H**	**H**	**C**	**H**	**C**	**H**	**H**	**C**	**C**
part7	**C**	**C**	**H**	**H**	**H**	**C**	**C**	**H**	**C**	**H**	**C**
part8	**C**	**H**	**H**	**H**	**C**	**H**	**C**	**H**	**C**	**C**	**C**
part9	**C**	**H**	**H**	**H**	**C**	**H**	**C**	**H**	**H**	**H**	**H**
part10	**C**	**H**	**H**	**C**	**C**	**C**	**H**	**C**	**H**	**C**	**C**

	1C	2H	3H	4H	5C	6C	7H	8H	9C	10H	11C
correct	**90%**	**50%**	**90%**	**90%**	**60%**	**50%**	**20%**	**70%**	**50%**	**30%**	**80%**
wrong	**10%**	**50%**	**10%**	**10%**	**40%**	**50%**	**80%**	**30%**	**50%**	**70%**	**20%**

Participants were 58.33% likely to correctly identify human produced poems on average. They were slightly more successful with my mechanical counterfeits with a success rate of 66.66%. Although it's a little self-defeating for a Turing test where

human productions are less recognizable than computer productions, I'm very impressed that they are within a 8% margin from one another.

Future Work:

Things I would like to improve upon in future editions of this project:

1- lines aren't generated independently of one another. That is, generated lines can exercise some influence on the creation of further content.

2- Improving POS tagging and syntactic glue. This ties in with point one, but even starting with the idea of making each line a grammatical clause could be a good starting point.

3- Using lines of alternate length to create more interesting forms.

4- Use of white space. Actually a very vital element of poetry in its visual appeal as well as a guide to the timing, phrasing, and cadence of phrases.

5- Experiment with using other training materials. I focused on rehashing poetry but I think it'd be just as interesting to make poetry from other sources.

If You Want to Play with my project:

open the file. There are three global variables, linelength, branch_factor, and poem_length you can adjust.

run main() which creates and returns an instance of the poetry model. catch it in variable and then call its method write_poemcouplets() or write_poemalt() to get an original poem. Again, if you find its taking to long, reduce the branch_fact, or linelength. Have fun!