Twitter Poetry Generator

Naif Alrayes
New York University
na1487@nyu.edu

Kwesi Daniel
New York University
kfd233@nyu.edu

Abstract

This paper discusses the development of a system that generates poetry based off of a corpus of Tweets obtained from Twitter. Twitter Poetry Generator generates poetry based on a user inputted search term. Our system searches twitter for about 10,000 words of a corpus, equivalent to about 1000 tweets, and creates poetry surrounding that specific topic. We use eSpeak to detect rhymes and count syllables; TweetNLP, a system devised at CMU that tags informal language based on Twitter; and TwitterSearch, a python wrapper for the twitter API, all to generate 4 line poems using a markov model and bigrams. The system successfully generate poetry, and informs the user when the search does not return enough results for it to generate rhyming words.

1 Introduction

The goal of our project was to devise a system that would generate 4-line poems from Tweets pulled from user-determined hashtags and search queries. Rather than using a single corpus to base our program off of, our system utilizes the Twitter API in order to pull the latest Tweets, making the corpus much more dynamic and representative of the current sentiments of Twitter users. As the grammar rules used to generate the poetry are also pulled from the same corpus, the poetry will incorporate the unique style of the text found on Twitter, as well as its flaws.

2 Poem Structure

The poetry our system generates comes in the form of 4-lines, each line containing 10 syllables. The final word in the first line will rhyme with the final word of the third line, with the second and fourth lines following a similar scheme.

3 Obtaining the Corpus

The first step in developing the system was to obtain the corpus to develop and test our system on. We used the TwitterSearch library (https://github.com/ckoepp/TwitterSearch) in order to utilize the features of the Twitter Search API. In order to create a brand new corpus upon each successive run, our system take in a query from the user and obtains the latest Tweets using the API, iterating through a certain number of tweets and holding them in a variable. We decided on 1000 tweets to be the size of our corpus since it gives us enough terms to have a well-rounded set of grammar rules and vocabulary. It also is a small enough number that we dont have to restrict our system to only popular queries containing tens of thousands of tweets.

4 Obtaining Pronunciation and Syllable Counts

Since we are working with rhymes, we needed a way to obtain the pronunciation of each word that we come across in the tweets. We initially considered a standard pronunciation dictionary, such as the CMU Pronunciation Dictionary. However, as Twitter is an incredibly informal platform, riddled with abbreviations, acronyms, slang, and an abundance of misspellings, we opted for another method of generating pronunciation that could handle anything we provide it, not just agreed-upon terms. For that, we discovered an independent project, Raplyzer, that incorporated the speech synthesis software eSpeak in order to generate pronunciation and rhymes. The eSpeak software has the capability of generating a pronunciation for almost any given term, including fictitious words. The software can also provide a text version of its pronunciation, complete with detailed phonemes for each part of the generated pronunciation. Through this, we were able to break the pronunciation into syllables, the last of which would be our rhyme scheme.

5 Algorithm for Rhymes and Syllables

To break up the word, we first needed a thorough understanding of how eSpeak works. In other words, what do the symbols that eSpeak returns as phenomes mean? Functionally, eSpeaks phenomes look a lot like the English language. As a result, our program really needed to just detect when any vowel sequences popped u p. The number of vowel sequences is the number of syllables. On top of that, if we divide by vowel sequences, we can detect whether a syllable is stressed or unstressed using the phenome that eSpeak returns, which could be useful for any future implementations involving iambic pentameter. Detecting rhymes at this point was easy, the only difference is that we needed to detect the last vowel sequence in the phenome, and be sure to include any consonants that came after those.

6 Tagging the Corpus

The next step was to tag the corpus. Using NLTKs built-in POS tagger was our initial idea, as it is a very effective and readily available system. However, NLTKs POS tagger was better designed for corpora that utilize proper grammar and commonly used words, the polar opposite of our corpus. Using NLTK on our corpus led to many terms with tags that were simply wrong. There was no way to properly handle all of the slang and abbreviations using NTLK. However, we did have an alternative. CMU has a POS tagger, TweetNLP, specifically designed to be used with tweets, which was exactly what we needed. The tradeoff, however, is that TweetNLP has fewer and less descriptive tags than NLTK and it has been less extensively developed. While TweetNLP does have the option to use PennTreebank-like tags, the accuracy of those was proven by CMUs statistical analysis to be far less than what has already been provided.

7 Generating Grammar Rules

After the corpus was tagged, we needed to generate the grammar rules in order to base our poetry off of. We settled on a bigram model which gives us enough rules and frequencies to work with considering the size of our corpus. A trigram model may have provided us with more meaningful and distinct grammar rules, but would require our system to generate a corpus several times larger than the current one, which in turn would prevent us from generating poetry from less popular search terms. The algorithm we used for the bigram model takes a tagged sentence in the form of an array of tuples containing the word and its POS tag. Before the first POS tag, however, the Begin tag is inserted so we can determine which tags tend to begin a sentence. We iterate through the list of tags for each sentence, appending the tags to a string. We simultaneously map each word to its respective POS tag in a dictionary with their occurrences. After iterating through each sentence, every tag is mapped in another to its preceding tag, followed by its number of occurrences. We use these occurrences to determine the probability that a word or tag follows another tag.

8 Generating The Poem

In order to generate the sentences, we first select a tag to begin the sentence. Based on their probabilities, we choose a weighted random tag start off the sentence and then generate a word that matches the tag, also based off of weighted random values. We keep track of the syllable count, making sure that it does not exceed 10 syllables per line. This process continues until we complete the sentence. However, if we come across a final tag that cannot lead to any words that match our remaining syllable count, then we regenerate the last tag and try again until we exhaust all of our possible ending tags. If that happens, then we toss the sentence. Generating a line that rhyme follows a similar process. However, we have another restriction. Most of the sentence is generated as before, except now, for each take rhyming into consideration. Each time we generate a new word, we decide whether we will have it rhyme or not, but the word must also complete the entire remaining syllables. A separate table contains all the possible rhyme groups, syllable counts, and words that match those syllable counts. When selecting a word, we will only look at those that both match the rhyme scheme and the remaining syllable count. If we are unable to generate a word that rhymes, then we follow a similar procedure to generating regular sentences and regenerate the entire tag. Once we generate the first two lines of the poem, the third and fourth need to rhyme with the previous line, generating our entire poem.

9 Baselines

In order for us to set a baseline for the sentence generation, we used an algorithm similar to the one we used for the poetry generation. We selected POS tags based on their weights until we reached the end tag. Since the baseline only generated random sentences, we did not need to take into account syllable counts and rhyming. The issue with generating until we reached the end tag is that often the end tag has a much lower frequency than the other POS tags, so the sentences the baseline algorithm generated ran on for multiple lines. However, it did prove as a way to determine whether our generating algorithm worked. To establish a decent proof-of-concept for our sentence generator while taking into account rhyming and syllable counting, we modified the above algorithm to count the number of words in the sentence. This proved as our syllable count. We then generated our first sentence. The second would then take the rhyme group of the last word in the previous sentence and use that as the rhyme group for the next. This assured that we could generate a pair of rhyming sentences that matched a 10 syllable count. In our actual algorithm, we replaced the sentence count with an actual syllable counter.

References: * TweetNLP: Archna Bhatia et al. Carnegie Mellon (http://www.cs.cmu.edu/ ark/TweetNLP/)
* Raplyzer Project: Eric Malmi (mining4meaning.com)
Twitter Search: Christian Koepp (https://github.com/ckoepp/TwitterSearch)
* N-grams and Markov chains: Allison Parrish (http://www.decontextualize.com/teaching/rwet/n-grams-and-markov-chains/)
* Language Technology Enables a Poetics of Interactive Generation: Antonio Roque Journal of Electronic Publishing Volume 14, Issue 2, Fall 2011 (http://quod.lib.umich.edu/j/jep/3336451.0014.209?view=text;rgn=main)
* Gnoetry: Eric Scovel (https://gnoetrydaily.wordpress.com)