

Final Project

Oh! A South Park Poetry Generator!

COSC-572 Empirical Methods in Natural Language Processing
Steven Chang (yc704), Tianrun Sun(ts1158), Yushi Zhao (yz521)

Abstract

In this project, we built a generator that teaches the South Park characters how to write poems in their own language, speaking style and mood. At the end of the training session, they were able to write poems to address their feelings to other characters.

1. Introduction

South Park is an American comedy set around four main characters known for their unique personalities. We believe that language use in the cartoon is an evidence of sharp character portrayals. We were curious if NLP techniques could be applied to create a language model that generates lines to reflect character personalities. In this project, we borrowed annotated data from Kaggle, and built an interactive poetry generator which features character selection, stanzas control, sentiment analysis and a rhyming system.

Watanabe et al. (2014) uses an n-gram model and a hidden markov model to build a generator for lyrics in Japanese. Besides, Maqsood (2015) compares and evaluates the ability to generate texts with sentiment using three generative models, latent Dirichlet allocation, Markov chains and hidden Markov Model. Oh and Rudnicky (2000) provides a good example of using vader_lexicon to apply sentiment analysis on Twitter data. As a result of all the previous work done on generation, we believed an n-gram model would be the most efficient to achieve a decent outcome. It involves lower computational complexity, which would give us time to implement more creative tasks such as sentiment analysis and poetry generation. At last, we added rhyme into poems, and meanwhile we also looked at Hayes (1998) which introduces rhyme systems in English poetry.

2. Dataset

Data download from Kaggle

Size: 5.28MB

Details: 71836 lines, annotated with season, episode and speaker Link:

<https://www.kaggle.com/tovarischsukhov/southparklines/data>

2.2. Understanding the dataset

The dataset was acquired from Kaggle, which has been annotated by seasons from the South Park animation. A sample of data(Fig. 1) is shown below with the attributes, “Season”, “Episode”, “Character”, and “Line”:

Season	Episode	Character	Line
10	1	Stan	You guys, you guys! Chef is going away.

Fig. 1

We took the data and dugged into the line of characters, since there are 3884 characters, the data is very sparse regarding of the lines from different characters. The chart below(Fig. 2) is the top 5 character with the record of “Number of Lines”, “Total Word Count”, “Average Sentence Length”, “Unique Words”, and “Lexical Diversity”, where Lexical Diversity indicates “Unique Words / Total Word Count”. In the lines Cartman owns the most lines, follow by Kyle and Butters. For the sentence length seems like the adult characters owns the longest sentences.

Name	Number of Lines	Total Word Count	Average Sentence Length	Unique Words	Lexical Diversity
Butters	2602.0	39729.0	15.27	3900.0	0.098
Cartman	9774.0	172646.0	17. 66	9602.0	0.056
Kyle	7099.0	85112.0	11.99	5316.0	0.062
Mr. Garrison	1002.0	19793.0	19.75	2507.0	0.127
Randy	2467.0	41595.0	16.86	3875.0	0.093

Fig. 2

In order to understand more of the dataset, the following plots(Fig. 3-6) shows the relation between different features in the chart above. Lexical Diversity and Total Word Count(Fig. 3) show a high density when the Total Word Count is relatively lower. Lexical Diversity drops

dramatically when the Total Word Count increases. This can be said as the main character who owns the most lines are prone to similar speaking style(using the similar words). As for the Fig. 4 a log-log plot indicate there is a negative trend between the two attributes.

As for the Fig. 5 and Fig. 6, is the relation between Unique Word Count and Total Word Count, the rate of increase is smoothly and slowly. This may because of the reason of most of the function words has been used and with only increase of the content words.

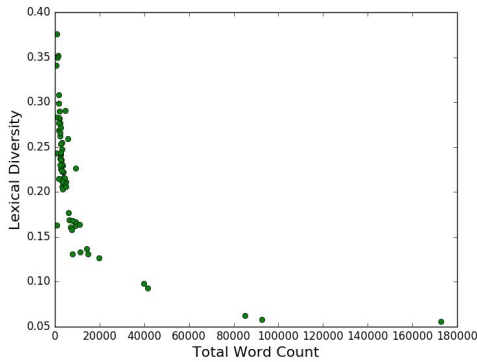


Fig. 3

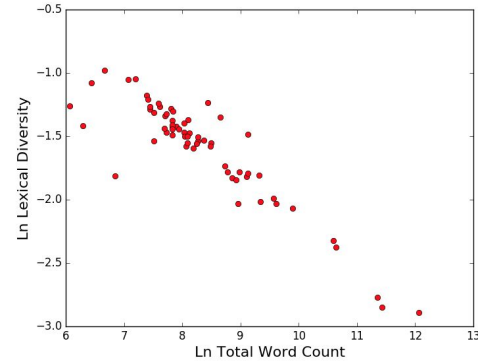


Fig. 4

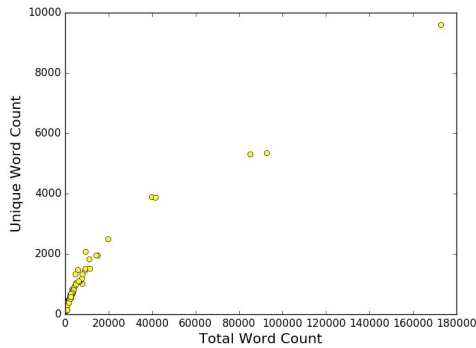


Fig. 5

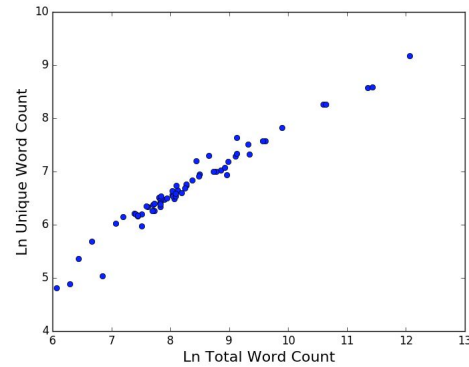


Fig. 6

3. Methodology

In this part, we will introduce the main method used to implement generation. N-gram is used as the language model and the generator is based on this language model. This section will cover how to set the sentiment values and make sentiment decision by “Vader_lexicon”.

3.1. Line Generator

From the previous dialogue mentioned, our goal is to make a generator sorted by characters. The first thing we need to do is building a language model or an n-gram model for the data. Also, it is essential to clean and format the data before that. Firstly, we divide lines and store them by characters. Then, those lines are tokenized with the help of function in NLTK. After that, the tokenized lines are able to be transformed into n-gram model and used for calculating sentence

probabilities. The intuition of generate a sentence is very naive.

Suppose we have a start word from the beginning of the project. It is possible to find the following words by scanning the list of n-grams starting with this word. Then, the last word in the current n-gram will become the start word of next n-gram. As a result, those n-grams will combine together and finally form a sentence. In order to choose the best fitting the following words in n-gram, the joint probability of each n-gram is compared with a defined threshold. If the joint probability is greater than the threshold, it will be appended into the sentence.

3.2. Sparsity

However, when we are running this generator model, we meet a problem that the dataset is very small, and after division by characters, data is more sparse. As a result, we cannot find a proper n-gram to build the lines. We took a different method to solve this problem. First of all, we define the value of threshold randomly. In the previous test, we set a concrete number for the threshold to filter those n-grams with a lower joint probability. However, no proper n-grams can be chosen. For example, in our dataset, there is a character named “GOD”, who has a few lines in the whole dataset.

Unfortunately, this person is often mentioned because many people said “oh god” or “oh my god” in the animations. As a result, this person has a higher frequency that our function would like to choose him as the “poem writer” while he cannot write it out because the joint probability of his lines is very small. A random threshold gives those n-grams a chance to continue generating lines even though there would be some grammar mistakes. The second method is to add a smoothing method for the data. In this project, we tried add-alpha smoothing and set alpha to 0.1.

Fortunately, it takes great efforts to solve sparsity problems. The last method aims at defining some personal rules for the data to resolve that problem. We have made some detection that most no result value problems occur in the end of a sentence. Since we have made a regulation that the last word of the sentence must be a terminated punctuation such as “.” “!” and “?”, these kinds of data are limited. For example, a line mentioning the character GOD looks like “Oh my god, Cartman!”. The end word should be Cartman. However, we cannot find an n-gram starting with X and ending in Cartman because this would cause deadlock in our algorithm. At last, we remove the restriction of the terminated punctuation and at the same time, we design a stopword to replace that.

3.3. N-gram

We have tried bi-gram, tri-gram and quad-gram as the language model. In our experiment, bi-gram has a best generating result. Since our goal is to build poems with short lines, if the length of n-gram is increasing, the sentences that are given back are more likely to be same. Thus, bi-gram shows the best result for this project.

4. Generator Features

4.1. Punctuation

Before calculating the sentence probability, we have to deal with the punctuation in the lines. The punctuation can be a mess, because the lines are annotated by human and the script is written to emphasize the mood or the environment of the animation. For example, the lines can be “How’s it goin’ ”, “WHAT?? ” or even “ A-are you okay?? ”.

Handling the punctuation can be difficult when dealing with the corpus of script, so we decided to substitute all the punctuation to “punc” after word tokenizing. While we set up some exception to keep the origin of the word, the hyphenate words and apostrophe s have been kept.

After generating the poem, the lines are full of “punc”. We transform the “punc” back to exclamation mark to fit into the strong and dramatic style of South Park. As for the implementation of the punctuation decision at the end of line will be discuss under the Stanzas Control.

4.2. Random Topic Generation

After the implementation of Line Generator, we have to come up with a idea of generating a poem’s topic, a name with random generation which is related to the dataset. The idea we come up with is let the user to input on of the main characters, and the topic generator will look through the lines of the character and then return a random selection of the character that have been frequently mentioned in the input character’s lines. For example, If the user input “Cartman”, the topic generator will randomly select from the top 5 character that Cartman mentioned the most, which are “[(745, 'Kyle'), (450, 'God'), (429, 'Butters'), (310, 'Kenny'), (275, 'Ho')]” (the number indicate the how many time the Cartman mentioned them.)

4.3. Stanzas Control

Now we have the topic of the poem, and the next step is to decide what kind format will be generated in our poem generator. After reviewing the paper *Quatrain Form in English Folk Verse*, we decided to use the Quatrain form poem which is a four line poem. In order to make the output poem be the format we wanted, user will be prompt to as ”How many stanzas for the poem?”, and the generator will based on the user input to generate the number of stanzas with the punctuation of “,” by the end of first three lines, and “.” at the last line.

4.4. Sentiment Control

South Park is an animation full with characteristic, every character have a very strong and unique way of speaking, nearly all of the topic of focus of the animation tends to be negative and ironic. By implementing the sentiment control of the system, we could have a better result and more interesting and uniform generation. The dictionary of sentiment score we use is Vader_lexicon which can be download and import by NLTK. Vader_lexicon is a function set in NLTK. It includes a dictionary that every word has a compared sentiment value in that.

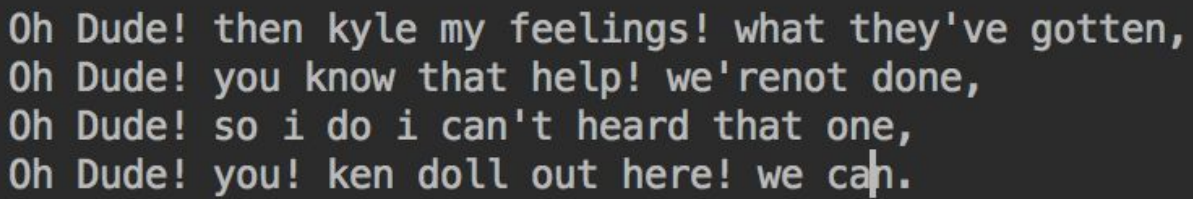
User can select sentiment type between “NEG, POS, WTV” which stands for negative, positive and whatever. For our input is a whole line. After training by this function, it return dictionary set of compound, negative, positive, and neutral value. Vader also sums all weighted scores to calculate a compound value normalized between -1 and 1. This value attempts to describe the overall effect of the entire text from strongly negative (-1) to strongly positive (1). The official Vader document suggests that a threshold of compound value should be -0.5 and 0.5. Which higher than upper bound(0.5) is considered as positive; lower than lower bound(-0.5) is negative. When choosing whatever, the score will be limited which mean no sentiment. We obey the threshold that the document defined and used it to judge the sentiment of the sentence.

4.5. Rhyme Selection

The goal is to have each line in a stanza end in the same rhyme. All pronunciations are provided by the CMU Pronouncing Dictionary. Essentially, two steps were involved in rhyme generation. First, we would have to ensure that the line ends in a non-OOV item in the CMU dictionary. Second, the rhyming word (which is defined by the last word in a line) must rhyme with the rhyming word of the first line in the stanza. If either step fails, the poem generator will regenerate new lines until the conditions are met. We incorporated two methods to determine rhyme, and they will be discussed in the following sections.

4.5.1 Rhyme by Final Syllable

In the first approach, rhyme is matched by extracting the V(C) cluster in the final syllable. For instance, the rhyme for ‘telephone’[T EH1 L AH0 F OW2 N] would be [OW N]. We would only extract the vowel if a word ends in an open syllable i.e. [K AE1 N D IY0] → [IY]. The numbers that denote stress are also deleted to ease the matching process. Note that many words may have multiple ways to pronounce, and we consider two words as a rhyming pair if at least one pronunciation matches.

A screenshot of a poem with four lines of text. The text is white on a dark background. The lines are: "Oh Dude! then kyle my feelings! what they've gotten,", "Oh Dude! you know that help! we're not done,", "Oh Dude! so i do i can't heard that one,", "Oh Dude! you! ken doll out here! we can". The words "gotten", "done", "one", and "can" are at the end of each line and rhyme.

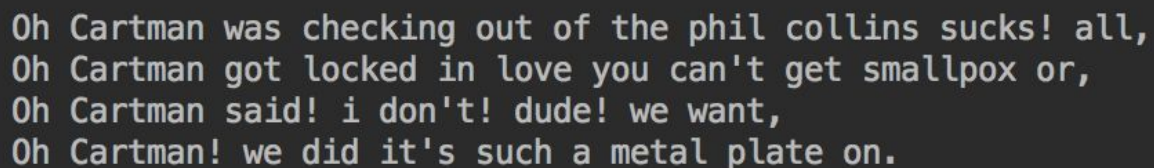
Oh Dude! then kyle my feelings! what they've gotten,
Oh Dude! you know that help! we're not done,
Oh Dude! so i do i can't heard that one,
Oh Dude! you! ken doll out here! we can.

Fig.7 An output poem with final-syllable rhyme

4.5.2 Rhyme by Stress

In the second approach, rhyme is defined by having the same vowels including and following the stressed vowel. Hence, we would only match [EH AH OW] for 'telephone'. Stress numbers and consonants would both be deleted. In cases where monosyllabic words do not carry a primary stress, the whole vowel string must be matched.

However, since usually the coda consonant is considered to be a part of the rhyme in a syllable, we sometimes get output words that are not strict rhyming pairs.

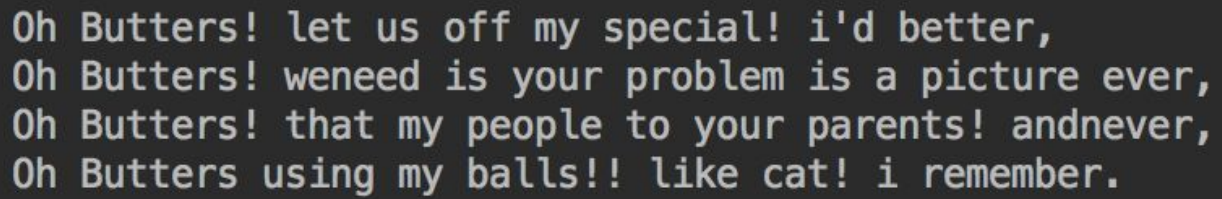
A screenshot of a poem with four lines of text. The text is white on a dark background. The lines are: "Oh Cartman was checking out of the phil collins sucks! all,", "Oh Cartman got locked in love you can't get smallpox or,", "Oh Cartman said! i don't! dude! we want,", "Oh Cartman! we did it's such a metal plate on.". The words "all", "or", "want", and "on" are at the end of each line and share the same vowel sound.

Oh Cartman was checking out of the phil collins sucks! all,
Oh Cartman got locked in love you can't get smallpox or,
Oh Cartman said! i don't! dude! we want,
Oh Cartman! we did it's such a metal plate on.

Fig. 8

'all', 'or', 'want' and 'on' share the same vowel ('want', 'or' and 'on' are expected to have multiple pronunciations); however, since coda consonants are not considered a part of the rhyme, the poem above does not sound so good. This is a remaining challenge for us because strings that denote a single phoneme are separated by a white space from the surrounding phonemes in the CMU dictionary, and therefore we couldn't use Regex to differentiate between onset consonants and coda consonants. Once we delete consonants, we have to delete all consonants.

However, In most cases, we get nice rhymes :)



Oh Butters! let us off my special! i'd better,
 Oh Butters! weneed is your problem is a picture ever,
 Oh Butters! that my people to your parents! andnever,
 Oh Butters using my balls!! like cat! i remember.

Fig.9

5. Input/Output

Input:

ONLY the character: Cartman, Kenny, Kyle,
 Stan

Who is the author:

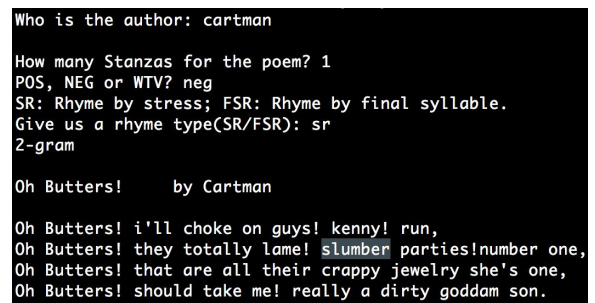
How many Stanzas for the poem?

POS, NEG or WTV?

SR: Rhyme by stress; FSR: Rhyme by final
 syllable.

Give us a rhyme type(SR/FSR):

Output: (Fig. 10)



Who is the author: cartman
 How many Stanzas for the poem? 1
 POS, NEG or WTV? neg
 SR: Rhyme by stress; FSR: Rhyme by final syllable.
 Give us a rhyme type(SR/FSR): sr
 2-gram
 Oh Butters! by Cartman
 Oh Butters! i'll choke on guys! kenny! run,
 Oh Butters! they totally lame! slumber parties!number one,
 Oh Butters! that are all their crappy jewelry she's one,
 Oh Butters! should take me! really a dirty goddam son.

6. Evaluation

The following Fig. 11 shows the evaluation of the result. Quad-gram has the best perplexity value while bi-gram is the worst one. However, bi-gram has a better generation result since the sentence length is very small, and if the n-gram size is very large, most parts of line would be covered by one n-gram and as a result, the sentences would look the same. Even though the bi-gram perplexity is larger, we still use bi-gram as the language model.

Perplexity	bi-gram	tri-gram	four-gram
1st line	7.7505	4.7748	3.6073
2nd line	6.8110	6.7973	4.0769
3rd line	5.0110	7.2724	4.4703
4th line	4.3866	6.0608	5.0054

Fig. 11

7. Challenges

Dealing with punctuation in the corpus is really a mess because the human annotators tried to be dramatic when transcribing lines. As mentioned above, we substituted all punctuation marks with ‘punc’, recovered all phrase-medial ‘punc’s with ‘!’s to mimic the dramaticness, and phrase-final ones with commas or periods. Even though we carefully observed the data, exceptions still pop up.

Furthermore, the output sometimes gives us the same rhyming word, and this may be due to the fact that we only allowed bi-grams that end in a punctuation to end our poem lines. This might give us a small bag of words from which we picked a rhyming word. Instead, we modified the model, and looked for all words including sentence-medial ones. However, the consequence of the modification is that a line could end in stop words or function words that left the phrase incomplete. We then created a short list of common stopwords that we often saw popping up in the output. On the other hand, we could also modify the algorithm so that the model keeps generating lines until a new rhyming word has been identified. Since the corpus size is not large enough to guarantee we always have at least three other words that match the rhyme in the first word, we occasionally ran into cases in which the generator could not find unique rhyming words.

8. Work Division

Steven: Model Generator, Stanzas and Topic generator, Sentiment generator

Yushi: Model Generator, Rhyme Selection

Tianrun: Model Generator, Evaluation

9. Github URL

https://github.com/s5745623/SouthPark_generator

References

- [1] Maqsud, Umar. (2015). Synthetic Text Generation for Sentiment Analysis. 156-161. 10.18653/v1/W15-2922.
- [2] Oh, A. H., & Rudnicky, A. I. (2000). Stochastic language generation for spoken dialogue systems. *ANLP/NAACL 2000 Workshop on Conversational Systems* -. doi:10.3115/1117562.1117568
- [3] Watanabe, K., Matsubayashi, Y., Inui, K., & Goto, M. (2014). Modeling structural topic transitions for automatic lyrics generation. In *Proceedings of the 28th Pacific Asia Conference on Language, Information and Computing*.
- [4] Hayes, Bruce P., and Margaret MacEachern. "Quatrain form in English folk verse." *Language* (1998): 473-507.
- [5] Loper, Edward, and Steven Bird. "NLTK: The natural language toolkit." *Proceedings of the ACL-02 Workshop on Effective tools and methodologies for teaching natural language processing and computational linguistics-Volume 1*. Association for Computational Linguistics, 2002.
- [6] Yuan, Bo. "Sentiment Analysis Of Twitter Data." *Rensselaer Polytechnic Institute, New York* (2016).