STACK ATTACK

A StackOverflow Markov Text Generator

COMPUTER SCIENCE 182: ARTIFICIAL INTELLIGENCE

Annie Lin & Joanne Koong

Introduction

We are interested in researching how to generate an artificially created but linguistically correct answer to a given tag or keyword.

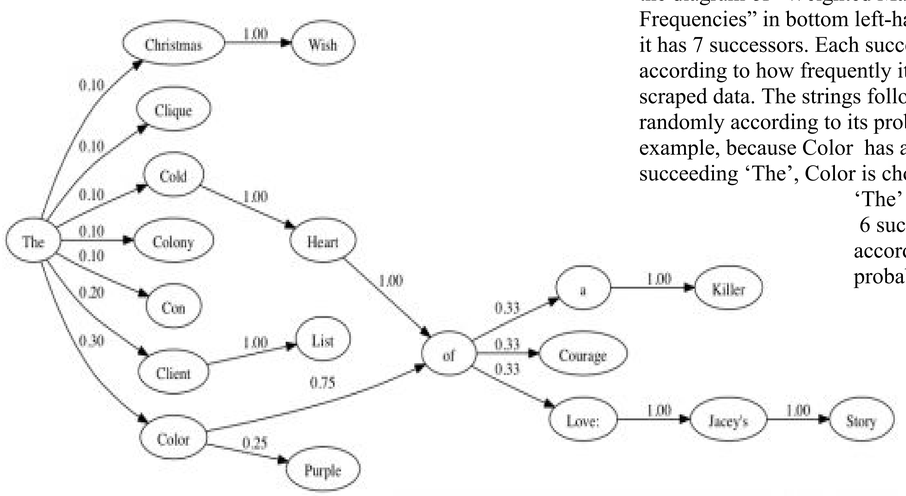
We used (the infamously famous) StackOverflow as our asking/answering platform for text generation using Markov processes.

Our StackOverflow answer generator prompts the user for a given number of sentences desired, and a pertinent tag ((http://stackoverflow.com/tags/synonyms) the user would like the generated answer to relate to. As explained more in-depth in *Methods*, we scrape the corresponding StackOverflow pages for the given user-inputted tag, and generate relevant "answers" that correspond to the tag.

For example, if the number of sentences desired was **3** and the tag was **'python'**, our text generator would output three sentences of helpful programming answers relating to the programming language python.

We decided to create a StackOverflow Markov Text Generator for our final project because we were intrigued by the probabilities underlying Markov chains (and its relation to training data), and weighting schemes (such as weighing different successive strings following a word differently depending on its frequency), and how we can utilize them in various domains, including language generation.

Weighted Markov Chain Using Word Frequencies



Methods

Scraping:

We use a third-party library, scrapy, to crawl across all the stackoverflow questions pertaining to a user-inputted tag. We then scrape the respective answers, and add them into an external data structure, a frequency dictionary that tracks the probability of word proceedings.

Search:

We first implemented our text generator using basic search. Our frequency "dictionary" is essentially one big array of all the words in the answers we have scraped.

We choose word proceedings randomly--we choose a random word from our file with no consideration to probabilities of it relating to its preceding word. Unsurprisingly, as indicated in results, our generated sentences do not make much sense.

Markov Chains:

We improved upon the *search* algorithm above by adding to our frequency dictionary a key, value pair corresponding to associations between successive words. In our dictionary, the key is the word in question and the value is an array containing strings that directly follow the word. We use an unweighted Markov chain where we do not take into consideration the different probabilities of strings that succeed a word--we merely collect all the different strings that follow the keyword and assign them equal probabilities of occurring (when in actuality, some string succeedings are more common for a given word than others).

Weighted Markov Chains:

We use the same algorithm as we did above for *Markov Chains* but with one important difference: we weigh the probabilities of succeeding strings according to how frequently it follows the word. In doing so, we are able to generate more sensical sentences. For example, let's refer to the diagram of "Weighted Markov Chains Using Word Frequencies" in bottom left-hand corner. For the word The, it has 7 successors. Each successor has a probability weight, according to how frequently it follows from the word in the scraped data. The strings following the word are chosen randomly according to its probability weight. In this example, because Color has a probability weight of 0.30 of succeeding 'The', Color is chosen as the word following 'The' 30% of the time, and the other 6 succeeding words are chosen according to their respective probabilities.

Results

```
Macintosh:hackharvard Annie$ python test.py
Number of sentences: 4
Tag (between quotes): 'python'
1. Search: append(sys. </p>\n\n<pre><code>&gt;&gt;&gt; data = eval(raw_input("En
ter a number: "))\nEnter a number: 5 + 17\n&gt;&gt;&gt; data, type(data)\n(22. &
lt;type \'int\'&gt;)\n</code></pre>\n\n<p><strong>Note:</strong> you should be c
areful when you are using <code>input</code> in Python 2. </p>\n\n<h3>How can <c
ode>__repr__</code> be useful?</h3>\n\n<p>Let\'s look at how useful it can be, u
sing the Python shell and <code>datetime</code> objects. </p>\n</blockquote>\n\n
<p>You can learn more about the other flags values that change / warn-about the
behavior of division by looking at the python man page.
Search Stats:
 Running Time: 2
 Readability Index: 23
 Word Count: 69
2. Markov: repeated times you need to compile but does not know what you success
ive chunks of using which is. elegance it in length if the function into one of
the package management alas for several classic python. continuations represent
it on the list of making a to make a windows sdk then. appear at instance of thi
s site the first if returns tuples and after making your code somelist for pytho
n.
Markov Stats:
 Running Time: 3
 Readability Index: 9
 Word Count: 70
3. Markov Weighting: records the algorithm that an iterable in name listvalue no
ne listvalue listvalue value your input almost always this working for. self has
 pointed by selective as nothing while r indices that shadows the op was. output
 got this x z for it introduces a great article on newer version would act a. cd
 da db support or y or try to immediately obvious and resolve the value then wou
ld advance.
Markov Weighting Stats:
 Running Time: 3
 Readability Index: 8
 Word Count: 70
```

We analyzed the performance and linguistic correctness of each method during our process. To test performance, we just measured the running time of each of the methods (using a simple time difference mechanism). To test linguistic correctness, we used the python library textstat, which measures the readability of English text. (The higher this number for an amount of text, the more readable the text is and presumable more sense the text makes.)

Also, to check consistency, we measured the word count when generating sentences: All of the methods generated around the same number of words for a given number of sentences.

We ran 100 iterations of our tests (which are encapsulated in one python script that you can run on the command line). An example output is included above, and below is a summary of our findings:

Search:

Because of our implementation, search took the least amount of time but also made the least amount of sense.

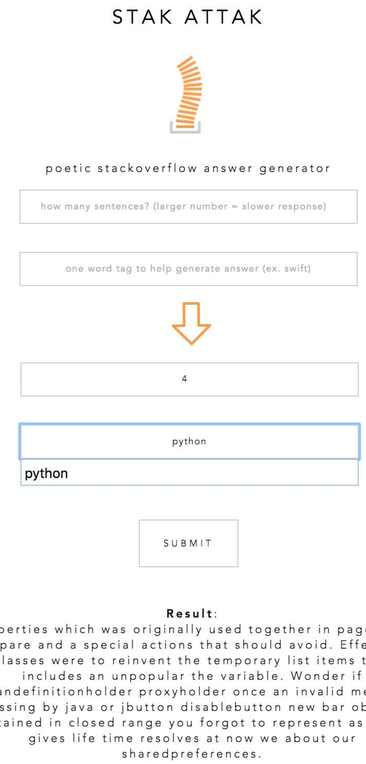
Markov Chains:

Using Markov chains dramatically increased the comprehensibility of our generated text, but, because we parsed more data, it took longer to run.

Weighted Markov Chains:

As expected, the readability of our generated text consistently increased from using just regular Markov chains.

Interface & Example



References

1. **Text Mining Using Markov Chains of Variable Length by Bjorn Hoffmeister & Thomas Zeugmann**

2. **Markov Logic by Pedro Domingos , Stanley Kok, Daniel Lowd, Hoifung Poon , Matthew Richardson , and Parag Singla**

CONTACT

Names:
Annie Lin & Joanne Koong

Emails:
annielin@college.harvard.edu
joannekoong@college.harvard.edu

QR Code