Bees. Thank you, like putting a crumb. - Where is on

Dustin Chung

Project: Jul 2018 — Writeup: Jul 2018

Overview

A simple Golang application was made to take a text input and to train a Markov chain on it. Then, the application is able to generate text based off of the training session. The training algorithim is kept extremely simple and the synthesis of words into sentences happens purely through probability, with no added logic for sentence structure, punctuation, or tokenization. The outputs of the program are entertaining if not incomprehensible. It should be stressed that this is a toy program, and will find little practical use in the real world in its current state.

1 Introduction

A Markov chain is "a stochastic model describing a sequence of possible events in which the probability of each event depends only on the state attained in the previous event".[1] Simply put, it is a process by which a system can change from one state to other states based off of some probability. This probability is also independant of any previous state of he process (i.e. it doesn't remember what it's done or the states it has been in previously). It has been well studied, and it's use for text generation well reported. A Markov chain was used to write the title of this writeup.

In this study, a Markov chain was used on strings of text. This is implemented by taking each individual word in the text, seeing what words come after it, and then recording them. A simple example is as follows. consider the following text:

```
"The apple is red."
      -> "the"   : "apple"
         "apple" : "is"
         "is"    : "red."
         "red."  : None
```

In this example, each word is unique. Any word in the sentence can only lead to one other word (with the exception of "red.", which leads to no other words since it is the last word). If we train the Markov chain on this sentence and tried to synthesize a new sentence starting with the first word, "The", we would only be able to generate the sentence "The

apple is red." Let us consider a slightly more exciting example:

```
"The apple is red. The banana is yellow."
      -> "The"    : "apple", "banana"
         "apple"  : "is"
         "is"     : "red.", "yellow."
         "red."   : "The"
         "banana": "is"
         "yellow": None
```

Here, we see that each time the word "The" appears in the text, the words "apple" or "banana" also appear and so are recorded. The same happens with the word "is": in the source text, the words that follow "is" are "red." or "yellow.". What does this mean? Well, now we have branching in our Markov chain which means that at the words "The" and "is", there is actually now more than one possible state to change into! If we let the chain pick its choice of word randomly, we can get a sizeable combination of possible sentences:

```
"The {F1} is {C1}. The {F2} is {C2}."

      Where F1, F2 can each be "apple" OR "banana",
            C1, C2 can each be "red" OR "yellow"
```

This is the basis of the super simple Markov chain used and talked about in this paper.

2 Methods

It is possible to tokenize and process the input text to a Markov chain in order to preseve attributes such as sentence structure and basic grammatical syntax. Techniques such as tokenization, or splitting the input text by a few words at a time instead of by one word at a time, are examples of this. However, the training algorithm used for the Markov chain in this paper is very simple and has no preprocessing. It can be expressed as follows:

1. Load the training text.
2. Create an empty map (in Golang: map[string][]string, in Python: dict).
3. Look at the first word of the text.
4. If this word is not already a key in the map created in Step 2, then create the key, point it to an empty slice/list and append the NEXT word (N+1) to the slice/list. If the word is already a key in the map, then just append the NEXT word (N+1) to that keys slice/list straight away.

5. Look at the next word of the text.

6. Go to step 4, repeat this process until there are no more words in the source text.

The code used for this training (implemented in Golang) is:

```
chain // empty map[string][]string
splitString // the source text, split into array of words

for index, item := range splitString {
  _, keyExists := chain[item]

  if !keyExists {
    chain[item] = []string{}
  }

  if index+1 < len(splitString) {
    chain[item] = append(chain[item], splitString[index+1])
  }
}
```

Once the chain has been trained on the text, there should be a map of strings to strings where the keys of the map are the individual words of the source text (with no repeats) and the values of the keys are a list of words that succeed the key word in the text. In other words, it should look a bit like the examples in the Introduction section. The benefit of doing things this way is that now, to generate new text from our trained chain, we only need to generate a random number lower than the length of the arrays in the map and use that number as the index for the respective keys' list. The benefit of doing it this way is that it makes stochastically generating a new string quite easy. There is no need to calculate any percentages. If there are two separate occurrences where the word "B" comes after the word "A", then the map will have recorded that as map["A"] = ["B", ..., "B"]. This means that when it comes time to make a new sentence from the trained chain, the relative probabilities of each successive word showing up are also recorded. In other words, in this example, the word "B" will be twice as likely to show up after "A" compared to another word that only succeeded "A" once in the whole text.

From a trained chain, generating new text is trivial. In this case, a random word from a list of the dictionary keys was chosen to be the first word of the new sentence. Then, it was just a matter of using a for loop to add new words as many times as was needed to generate text of a desired length.

```
keys := make([]string, len(chain))

        i := 0
```

```
for item := range chain {
        keys[i] = item
        i++
}

output := []string{keys[rand.Intn(len(keys))]}

for wordInStory := 1; wordInStory < length; wordInStory++ {
        lastWord := output[len(output)-1]
        var nextWord string

        if len(chain[lastWord]) > 0 {
                nextWord =
chain[lastWord][rand.Intn(len(chain[lastWord]))]
        } else {
                nextWord = keys[rand.Intn(len(keys))]
        }

        output = append(output, nextWord)
}

output[0] = strings.Title(output[0])
return strings.Join(output, " ")
```

Note the line "output[0] = strings.Title(output[0])", which just capitalizes the first letter of the first word, and "nextWord = keys[rand.Intn(len(keys))]", which picks a new random word to continue the text in case the Markov chain has hit a "dead end" (has no more words to choose from to continue).

3 Results & Discussion

The generated texts speak for themselves. With a purely stochastic technique, grammatical errors were bound to happen. Even so, some of the outputs were entertaining to read and were even quite characteristic of the sources in which they were trained on. It goes without saying that the more words the chain is trained on, the more "correct" they will sound. The following are FIRST TIME generated texts (no cherry picking results). The generated output length was capped at 30 words. With no mechanism to make sure that sentences ended on fullstops, there are some hanging lines.

This first one is from the Loreum Ipsum. I don't know this language so I can't really tell if it's right or wrong. Bad for analysis, good for self-esteem seeing how the other examples turned out though.

```
Donec faucibus imperdiet dui sagittis gravida tellus et
malesuada a. Aliquam congue ipsum quis neque vehicula
ex efficitur. In blandit felis. In blandit sodales
urna euismod, sed turpis

– 5 generated of Loreum Ipsum from https://lipsum.com/
  Source word count: 440, unique keys: 219.
```

The next one is text generated from a chain that was trained on an excerpt of a writeup I did for one of my other projects, pastebin_buddy. The funny thing is, I didn't realize there was a typo in that writeup until the generated text showed it (accouts).

```
Once a PB server like the landscape of the users
(Pro). For example, Free accouts have a link to
be delays in question were gaining their follower
base thorough illegitimate

– Excerpts from my pastebin_buddy project writeup!
  Source word count: 440, unique keys: 249.
```

Here, it's been trained on song lyrics. More specifically, "We Didn't Start the Fire". It's a song with a fair bit of words, most of which are unique (there are lots of names in this song), so the generated text has a very linear progression. Towards the end it even repeats the beginning of the chorus word for word because of this.

```
Brando, "The Catcher in the shores, China's under
 martial law Rock and Prokofiev Rockefeller,
 Campanella, Communist Bloc We didn't start
 the fire It was always burning Since the world's

– Lyrics from "We Didn't Start the Fire" by Billy Joel
  Source word count: 497, unique keys: 254.
```

This one makes the most sense out of the previous examples. The chain has been trained on the entire script of The Bee Movie. The source text was not cleaned up beforehand, leaving in long spaces and hyphens where line breaks and the actor names were supposed to go. Strangely enough, the line breaks were not caught by the formatting code used to take out newline and tabspace characters, hence their appearance in the text.

```
Lawsuit'S a minute. I never sue humans.
– Where is on our honey? That's pollen power.
 Ready, boys? The bees! They
```

```
– Entire script of The Bee Movie
  Source word count: 14654, unique keys: 2928.
```

It can be seen that when the ratio of source word count to unique keys increases (i.e. there are a lot more non-unique, recurring words than repeated words), the quality of the sentences in terms of grammar and readability also goes up. This is because the abundance of repeated words gives the Markov chain more "freedom" to choose the next word at any particular point and so the generated text does not become too linear, or non-sensical, since theres is a lower chance of having to choose a new random word to restart the sentence when it hits a dead end. To test whether this explanation was true, the chain was run one last time, this time tested on one of the Winnie the Pooh books. A childrens book was chosen because it would have relatively simple English, so not too many unique words would be recorded. Winnie the Pooh books are also fairly long, allowing for a high source word count to unique key ratio:

```
Paints to yourself, Little Piglet?" said Pooh.
 "Thank you, just before that she got a starling
  or Pooh couldn't see if

– One of the Winnie the Pooh books.
Source word count: 32141, unique keys: 3930.
```

4 Conclusion

Using simple stochastic Markov chains to generate text with no special preprocessing gives only barely passable results, although the generated texts do get better as the ratio of the source word count to unique keys increases. The chains could possibly be improved a lot by preprocessing the data (tokenizing, splitting by groups of 3 words instead of each word), though a neural network might be able to generate better results if given texts and labels which categorize the text as understandable or not. The use of natural language processing libraries and resources will also have marginal effects on the output, depending on how they are used, but is beyond the scope of what was set out to be accomplished.

References

[1] Oxford Dictionaries; https://tinyurl.com/y7qqkskk