Computational Creativity Report

Jahnvi Sikligar, 210538601
School of Electronic Engineering and Computer Science,
Queen Mary University of London, UK

ec21204@qmul.ac.uk

The Markov chains have proven to be extremely useful in the development of most AI text generators. Inspired by this creation, I attempted to implement a creative text generation system that uses the Markov process and N-Grams techniques to generate short stories.

The system will compose a prose with short paragraph based on the work of a well-known author. The most obvious method is to examine the novel to see how the words flow together. Then, given the N previous words, it would determine a likely 'new' word to follow.

1 Introduction

The Artificial Intelligence has crept its way into novel writing and text generator. Jasper.AI (Conversion.ai) is one of the most prominent examples that generates stories and novels using artificial intelligence. Thousands of writers recommend and use the tool to write entire blog posts. It makes use of OpenAI's advanced GPT-3 algorithms to generate unique and compiling storyline text based on the text you supply. This tool can also be used to create other types of content, such as blog posts, copywriting ads, newsletters, product descriptions, and so on. It understands what you want it to do better than any other tool on the market, so you can trust that its output will always be of high quality.

The Markov chains have also proven to be extremely instrumental in the development of most AI text generators. Inspired by this creation, I have intended to implement a creative system for writing short prose with the help of Markov process and N-grams. The idea behind the project is specifically designed for text generation domain where the creative system generates prose whose style of writing is similar to the original author's writing as observed in a particular novel or book.

With this thought process to recreate one particular author's style. We will be using Markov chains and the N-gram technique of grouping adjacent words together (where N is the number of words in each group).

2 Background

If we carefully read and observe writings of various authors, it could be seen that each author has their own style of writing and are confined to a particular genre. One such example is, G.B.Shaw whose plays had a peculiar style of Shavian attitude. His work is often comedic and clever, which he was careful to do in order to disguise his message as entertaining and enthralling. To recreate the style of a specific author, the system uses an N-gram technique (where N is the number of words in each group) that will group adjacent words together using Markov chains.

As we know, an N-gram is a contiguous sequence of n text items from a given text sequence. We can create a list of n-grams from a sentence, s, by finding pairs of words that occur next to each other. To find the probability of the entire sentence, simply look up the probabilities of each component part in the conditional probability. Unfortunately, because we cannot compute n-grams of any length, this formula does not scale.

In order to reduce such complexity, the Markov Property is a popularly used with n-gram modelling. According to the Markov Property, the probability of future states is determined solely by the current state and not by the sequence of events that preceded it. This concept can be elegantly implemented by storing the probabilities of transitioning to the next state in a Markov Chain.

3 System Description

The text generative system is divided into two parts where it initially tries to analyse the next possible word with the computation of position transition using Markov chains.

As discussed earlier that we are trying to examine the writing style of an author to recreate prose or short paragraphs that are similar to his work. For this project we will be using data in form of a text file – **the_time_machine.txt** of the novel – **The Time Machine** written by renowned author for fictional genre H.G. Wells. The text data file has been taken from a public open source Project Gutenberg. The link for which has been provided in the Colab file. It can be noticed that there has been changes made to the original .txt file so as to avoid any irrelevant sentences creation at the very end of this project.

Firstly we will start by loading the data into the Colab for it to be examined for future purposes in the whole project. The entire **the_time_machine.txt** is stored in the string **the_time_machine**. Then we start with understanding the text file, so we

- print the first 750 characters of the novel by using the slicing method as shown in the code.
- Check the number of characters in novel using len() function
- using 'split' method to compute the actual word count by removing whitespaces(new line, tab, space, etc..).
- return a Python list of the first 50 words of the novel like this (notice how I'm using "slicing" again):

Our main objective is to generate prose that's based on H.G.Wells. The most obvious way to analyse the novel is to look at how the words flow together. If we could figure that out, we could write some code that, given N previous words, would figure out a likely new word to follow.

The N-gram is a technique for grouping adjacent words together (where N is the number of words in each group). It's the kind of thing that's obvious but may be difficult to explain... so here's an example. Following output can be estimated for:

N = 3,	N = 2,
"The Time Machine" "An Invention by"	"The Time" "Machine An" "Invention by"

We will be using python's builtin zip function. The zip function will take in a list of iterables and create a new list of tuples where each list will contain inputs of the elements accordingly. The zip function has been explained in detail in this blog post: (http://locallyoptimal.com/blog/2013/01/20/elegant-n-gram-generation-in-python/).

We already have the list 'all the words in the novel' that should be passed into the function. The second argument, 'n' should be the n-N gram's value. The end result will be a collection of all the n-grams from The Time Machine. Let's begin with N = 3. If we choose an starting n-gram at random, then just keep generating a new word some arbitrary number of times. The output will start with the three words of the first (randomly selected) n-gram and we just need to append each new word until we tell it to stop.

Let’s consider random examples, we have a generated n-gram "it is a" and we choose "most" at random as the next word in our generated text; the new n-gram is "is a most." Given the contents of the preceding example sentences, the candidate pool of next words to follow the "is a most" n-gram includes the words "vexing" and "wonderful." This process will continue till we decide to stop it. This is essentially how a Markov chain works to generate new superficially real-looking output based solely on the n-gram input data. Here we create a list of following words for each n-gram. This is easy to achieve with a Python dictionary: if the key is the n-gram, then the associated value can be a list of all the words that ever followed the n-gram in the key. The simplest way to do this is to grab n-grams of N+1 to get the n-grams of length N and the following word (N+1).

At this stage, the skeleton of what we need to generate H.G.Wells like prose is ready and can be used to check if it gives expected output or not. It can be done if we choose an starting n-gram at random, then just keep generating a new word some arbitrary number of times. The output will start with the three words of the first (randomly selected) n-gram and we just need to append each new word until we tell it to stop.

A method for producing syntactically correct sentences (i.e. they start with a capital letter, end with the correct punctuation and any quotation marks are correctly balanced). This is where the real challenge begins, because we can programme some heuristics (general rules of thumb) to assist us in overcoming these problems. For example, how can we ensure that all sentences begin correctly...? Let's just make a list of all the n-grams that contain words that started a sentence in the original text.

Getting the `open_n_grams` involves several considerations:

- We need to identify detect when there's the end of a sentence (so the following words will become part of an opening n-gram). We detect by matching against common `stop_chars` which indicate the end of a sentence.
- The start of a sentence must begin with a capitalised letter or an opening quotation mark.

The actual process of creating opening n-grams of length N, actually involves using n-grams of N+1. Say N is 3 then we need all the 4 word n-grams in the novel so we can act on them like this:

- Imagine a 4 word n-gram: `carefully. I shall have` (taken from the `the_time_machine.txt` file)
- Firstly, checking the first word `carefully.` has a final character which is a stop character(it does).
- Second, check it's not in the tricky words to ignore (it's not).
- Third, so far so good, so check the second word to see if it either starts with a capital letter (it does not) or a quotation mark (it does).
- Fourth, we've found an n-gram that we can use! So, remove the first word (`carefully.`) leaving us with an n-gram of length N: `I shall have`
- Add this n-gram to the `open_n_grams` list.

If any of the various checks described above are unsuccessful, then that particular n-gram is ignored and we move onto the next N+1 n-gram. To generate a new seed for starting a sentence we can just use `random.choice` on the `open_n_grams` list to get an appropriate n-gram. Next step involves creation of sentences for which we have defined ‘makesentence’ function which takes in desired ‘word_length’ as a parameter. The ‘makesentence’ paragraph is revised with a condition to keep on running till it is able to find words that make the sentence syntactically correct. Later on flagging can be done to remove words that end with quotation marks to revise candidate list.

While we worked out to closely generate sentences correctly, next step is to implement paragraph and chapter generation of text. Three new functions, as well as changes to the existing ‘makesentence’, are defined in the following code:

- **makenewseed** - This function takes the n-gram at the end of the previous sentence and generates an n-gram to begin the next new sentence. This directly addresses the requirement that the preceding text's final n-gram serve as the seed for the text that follows. The important property of the result is that it will be a seed n-gram that starts a new sentence correctly.
- **makesentence** - This function works exactly like the previous one, except it takes a seed n-gram as an argument and returns two values as a result: the new sentence and the final n-gram used to generate the sentence. As a result, we can chain these functions together to generate the seed n-gram for the next sentence using the final n-gram passed through make new seed.
- **makeparagraph and makechapter** - These two functions operate in a very similar manner. You give them a number to indicate how much of what they produce you require, as well as a seed n-gram to get them started. They both have a loop that will run for however long you specify to generate content before returning their result.

Lastly we come to the end and main objective of this project where as an output short 7 paragraph chapter is generated using a seed n-gram that contains the opening three words of the novel.

4 Experiments and Results

After the first step of loading the data – **the_time_machine.txt,** the analysis of the data is done in following steps:

- Printing of the first 750 characters of the novel

```
print(the_time_machine[:750])  # Print the first 750 characters of the novel.
```

```
The Time Machine

An Invention

by H. G. Wells

 I.
 Introduction


The Time Traveller (for so it will be convenient to speak of him) was
expounding a recondite matter to us. His pale grey eyes shone and
twinkled, and his usually pale face was flushed and animated. The fire
burnt brightly, and the soft radiance of the incandescent lights in the
lilies of silver caught the bubbles that flashed and passed in our
glasses. Our chairs, being his patents, embraced and caressed us rather
than submitted to be sat upon, and there was that luxurious
after-dinner atmosphere, when thought runs gracefully free of the
trammels of precision. And he put it to us in this way—marking the
points with a lean forefinger—as we sat and lazily admired his
earnest
```

- Check the number of characters in novel using len() function

```
len(the_time_machine)  # to display number of characters (as in letters in the novel)
```

```
179298
```

- using 'split' method to compute the actual word count by removing whitespaces(new line, tab, space, etc..).

```
[61] all_the_words_in_the_novel = the_time_machine.split()  # Split the novel by whitespace.
     len(all_the_words_in_the_novel)  # The length of all_the_words_in_the_novel
```

```
32385
```

- return a Python list of the first 50 words of the novel:

```
['\ufeffThe',
 'Time',
 'Machine',
 'An',
 'Invention',
 'by',
 'H.',
 'G.',
 'Wells',
 'I.',
 'Introduction',
 'The',
 'Time',
 'Traveller',
 '(for',
 'so',
 'it',
 'will',
 'be',
 'convenient',
 'to',
 'speak',
 'of',
 'him)',
 'was',
 'expounding',
 'a',
 'recondite',
 'matter',
 'to',
 'us.',
 'His',
 'pale',
 'grey',
 'eyes',
 'shone',
 'and',
 'twinkled,',
 'and',
 'his',
 'usually',
 'pale',
 'face',
 'was',
 'flushed',
 'and',
 'animated.',
 'The',
 'fire',
 'burnt']
```

The zip function will take in a list of iterables and create a new list of tuples where each list will contain inputs of the elements accordingly. The end result will be a collection of all the n-grams from The Time Machine. Let's begin with N = 3:

```
[('\ufeffThe', 'Time', 'Machine'),
 ('Time', 'Machine', 'An'),
 ('Machine', 'An', 'Invention'),
 ('An', 'Invention', 'by'),
 ('Invention', 'by', 'H.'),
 ('by', 'H.', 'G.')]
```

Given a low value of N (say 3) we'll find that the same three word n-gram will appear several times in the work. However, each time it may be followed by different words. Of course, perhaps the same word will follow the n-gram several times. If we choose a "opening" n-gram as a seed for generating H.G.Wells -

style prose, we can work out a candidate for the next word in the sentence by randomly selecting one of the words in the list of words that follow that n-gram.

To begin, make a list of the following words for each n-gram. With a Python dictionary, this is simple: if the key is the n-gram, the associated value can be a list of all the words that ever followed the n-gram in the key. This should include repetitions as well. The simplest method is to take n-grams of N+1 to get n-grams of length N and the following word (N+1).

```
'it grip me at the throat and stop my breathing. In another moment I was in doubt of my direction. I looked into the glaring eyeballs. I was afraid to turn. Then the thought of'
```

What we really need is a way to generate syntactically correct sentences (i.e. they start with a capital letter, end with the correct punctuation and any quotation marks are correctly balanced). Let's just make a list of all the n-grams in the original text that contain words that started a sentence:

```
[('His', 'pale', 'grey'),
 ('The', 'fire', 'burnt'),
 ('Our', 'chairs,', 'being'),
 ('And', 'he', 'put'),
 ('I', 'shall', 'have'),
 ('The', 'geometry,', 'for'),
 ('You', 'will', 'soon'),
 ('You', 'know', 'of'),
 ('They', 'taught', 'you'),
 ('Neither', 'has', 'a')]
```

After this next step is sentence creation. Initially a function 'make sentence' is written that accepts a word-length parameter to check its basic working. Later on we enhance the same function so to get syntactically correct sentence by limiting the candidate words to the ones that end in stopping characters.

checking the basic version working:

```
[ ]  1  ' '.join(makesentence(8))

'I could see the many palps of its'
```

Output of ‘makesentence’ basic version

If we try this final revision I think we're close enough (but certainly not pe correctly.

```
[63]  1  ' '.join(makesentence(8))

'He hesitated. His eye wandered about the room.'
```

Output of ‘makesentence’ final version

For the final stage we have defined functions ‘makechapter’ , ‘makeparagraph’ and ‘makenewseed’ and ‘makesentence’. The ‘makechapter’ re-uses ‘makeparagraph’ which re-uses ‘makenewseed’ and ‘makesentence’. The output of the same is as shown below:

```
The Time Machine An Invention by H. G. Wells I. Introduction The Time Traveller hesitated. Then suddenly: “Certainly not.” “Where did you really get them?” said the Me
“The building had a huge entry, and was altogether of colossal dimensions. I was naturally most occupied with the growing crowd of little people, and with the same pec
They still possessed the earth on sufferance: since the Morlocks, subterranean for innumerable generations, had come at last to find the daylit surface intolerable. An
Catching myself at that, I took my eyes off the Time Traveller’s face. IV. Time Travelling “I told some of you last Thursday of the principles of the Time Machine. Tow
In manœuvring with my matches and my camphor I could contrive to keep my path illuminated through the woods. Yet it was evident that if I was to appreciate how far it
A minute passed. Their voices seemed to rise to a higher pitch of excitement, and their movements grew faster. Yet none came within reach. I stood glaring at the black
“At once, like a lash across the face, came the possibility of losing my own age, of being left helpless in this strange new world. The bare thought of it all. “It sou
```

5 Discussion

At the stage where he had the skeletons for the text generation in the form of prose the output still generated new arbitrary words when it was executed. It could be seen that are re-running the code the resulting output still had flaws:

- typically begins in the middle of a sentence.
- abruptly terminates mid-sentence.
- could have an unopened or unclosed quotation mark.

But it did not meet the requirement of producing sentences which were grammatically correct. This was a major challenge. To get better understanding on how the author began his sentences 'open_n_grams' list was created and values were added accordingly. We were still left to know if the sentences had correct punctuations('.!?') and quotation marks. For this basic version of 'makesentence' function was implemented. However, we needed to examine if the sentence was grammatically correct or not. To implement this the candidate words were limited to only those with **stop characters**. In the final version of 'makesentence' it could be seen that 'make_next_word' function was created to throw away words that ended with quotation marks. It could be seen that final revision was close enough in making sentences that were nearly correct.

6 Conclusions and Future Work

While the generative system had obtained results of creating prose that was quite close to the writing style of author H.G.Wells. The system can also be tested on the various other books, novels and short stories too. But it still had scope for improvisations in it. As noticed the experimentation was done in smaller fragments to try and generate the best results. The system did understand the stopping characters, opening words and ending words from the text file. It was still speculating the correct usage of grammar which could be a point of improvisation. Also, it has been able to implement only prose style text but different authors have varied ways of writing.

References

1. https://aircconline.com/ijaia/V10N6/10619ijaia04.pdf
2. Gagniuc, Paul A. (2017). Markov Chains: From Theory to Implementation and Experimentation. USA, NJ: John Wiley & Sons. pp. 1–235. ISBN 978-1-119-38755-8.
3. https://www.jasper.ai/
4. https://sookocheff.com/post/nlp/ngram-modeling-with-markov-chains/#&gid=1&pid=1
5. https://sookocheff.com/post/nlp/n-gram-modeling/
6. https://www.rednoise.org/pdal/index.php?n=Main.N-Grams
7. https://analyticsindiamag.com/hands-on-guide-to-markov-chain-for-text-generation/
8. https://setosa.io/ev/markov-chains/