Final Project for CS 372

Jonathan Vottero

Running the code

The code works best in a linux environment, but I did get it to run through pycharm in windows. To run the code in linux navigate to the file where the code is and type ./Markov_Chain.py

Algorithm, Application, Language Choice

- Markov Chains
- Text generation
- python

Where It Is Used

My algorithm is used to model states and the transitions between them. Like the definition for the algorithm the applications for this algorithm are very broad. It can be used to predict the probability of a state transition based on a current state or it can be used to simply analyze the connections of many different states. In physics a Markov chain can be used to predict the next state of a system whenever probabilities are used to represent unknown or unmodelled details of the system. Google used a Markov chain in their page rank algorithm in order to determine the strengths of connections of between different webpages.

For text generation Markov chains are not the most correct solution, if you are trying to create a convincing sentence. However, I chose to use a Markov chain because it is simple to code and has a fast runtime. For more human like sounding text you could use a neural network, trained on a large sample set of text. The downside to this approach is the large processing power/training time needed in order to get a well-trained network. Other types of text generation rely heavily on the context in which the text is being used. For example, a data base interpreter would rely on a network of logic checks related to the data in the database to provide the user with data that was human readable. This approach would also get you more "human like" readability with comparable runtimes but would require much greater knowledge of the system you are trying to generate text for.

Validate Theoretical Correctness

The algorithm works by first parsing a text file into a list of words and symbols in the order that they appear in the text. This step should get rid of any uncommon punctation and help us format the data for the next step. Markov chains are usually represented by a graph where each node represents a state and each weighted edge represents the probability of the transition from the current state to the connected state. For text generation we will consider or current state to be the current word we are looking at and the next state to be the word we will likely see next in the sentence. The probability of going from our current word to the next is the

number of times we see this word directly followed by the word we are evaluating divided by the amount of times we have seen our current word. An example would be: If we are currently looking at the word “united” we could see that “united” was found in our text 100 time and 90 of those times it was followed by the word “states”. So, if we see the word “united” we know that there is a %90 chance that the next word is “states”. To create this graph structure of words and their probabilities we need a 2d array where our rows are indexed by the current word we are on and our columns contain all the words that have followed our current word along with the number of times we have seen them follow this word. We will also want to keep track of the total amount of times we have seen the current word, so we can calculate the probability of each state transition.

My program does this by walking through our list getting the current word and the word after it and creating this Markov chain graph representation. Then to generate a sentence simply pick a starting word to use as the graph start node and follow the edges by randomly selecting the next node based off the stored probabilities. You can keep going until you hit ending punctuation and bam you have a random sentence!

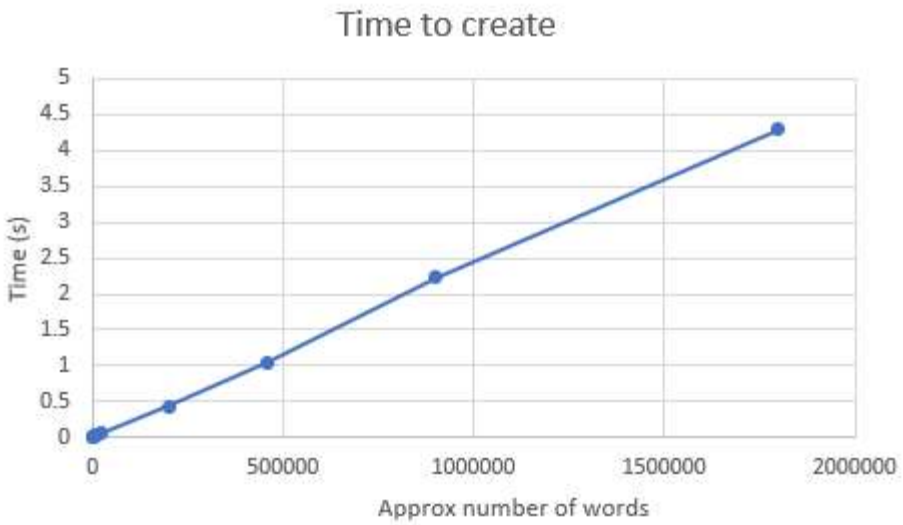
Run time

For the runtime I was unsure if you would want an analysis of the creation of the Markov chain or the use of the Markov chain to generate text, so I did both. For simplicity I use the number of words as my n. In reality it is the number of words and valid punctuation, like commas and sentence ending punctation. This does not effect the run time analysis much however because the number of occurrences of those symbols should scale linearly with the number of words. You can also run the time tests yourself through the command line interface.

Creation: This is the basic core of the algorithm used to create the Markov chain.

```
wordlist = paseWordList(filename)                <- O(n)
foreach i in length of wordlist:                 <- O(n)
        word = wordlist[i]                       <- O(1)
        nextWord = wordlist[i+1]                 <- O(1)
        markovMatrix[word][nextWord] += 1        <- O(1)
        sums[word] += 1                          <- O(1)
```

Overall it should be O(n). We see this In the runtime test.

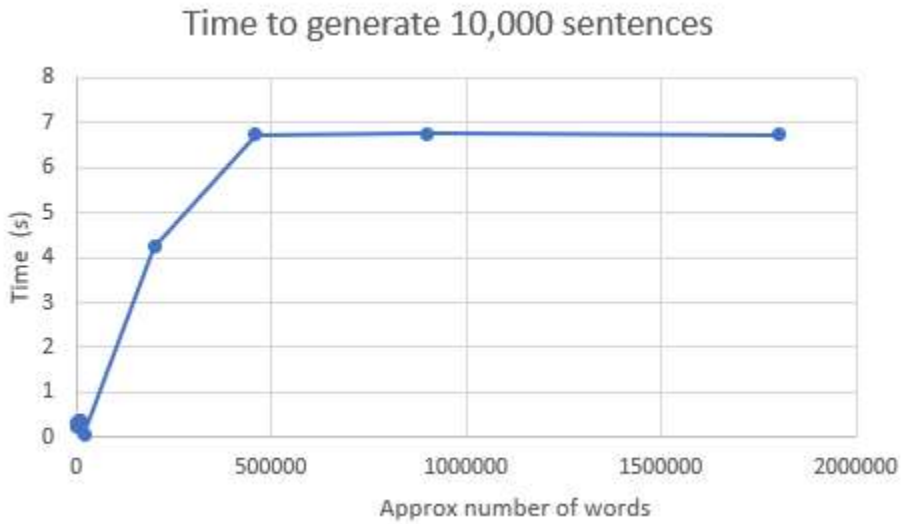


Word Generation: With the implementation that I used this step should take O(k*s) time where k is the average number of connections for each node and s is the average length of a sentence. Using a more advanced method you could get constant time for the weighted traversal which would just give you O(s*1) time or just O(1) time, because s is a constant across most English text. My pseudo code for a single word look up is below:

```
wordRow = markovMatrix[word]
total = sums[word]
randNum = random.randint(1,total)
    for word in wordRow:                    <-  O(k/2) time on average
        randNum -= wordRow[word]
        if randNum <= 0:
            return word
```

Final run time for one word: O(k)

We can see how the run time is affected by the graph connections in the runtime graph below.



In the graph you can see the linear increase in time as the files get bigger, but it then flattens out after 50,000 words. This is because the text for the 90,000 and 180,000 word files is simply the text from the 50,000 word file copied and pasted. This caused the graphs for each file to look very similar and caused k to be roughly the same for each file.

Built-in Code Correctness Tests

All tests can be run through the console interface and their pass/fail status will be output to the console.

Test 1; regEx tests

The first set of tests ensures that the parsing happened correctly.

Input

regExTest.txt

Output

A word/symbol list of the parsed file.

Test 2; Markov matrix generation

These tests ensure that the Markov matrix was created properly.

Input

genMarkovTests.txt

Output

A Markov 'array' with all the words as rows and the probability of state transitions as columns.

Test 3; Sentence generation

These tests ensure that the Markov matrix is used correctly to generate sentences.

Input

sentenceGenTests.txt – a file with were all Markov graph edges have a probability of 100%

Output

Four sentences, each corresponding to a sentience in the original file.

Test 4; Combining Files

These tests ensure that the program can combine the data from two files into one Markov matrix

Input

combineTest1.txt and combineTest2.txt

Output

A Markov matrix that takes into account probabilities and sums from both files.

References

Explanation video:

https://www.youtube.com/watch?v=56mGTszb_iM

Wikipedia articles:

https://en.wikipedia.org/wiki/Markov_chain

https://en.wikipedia.org/wiki/Natural-language_generation