Lab 9: Review and Markov

CSE/IT 107

NMT Department of Computer Science and Engineering

"Holy shit, you geeks are badass."

— Pam (*Archer*)

"Simplicity is prerequisite for reliability."

— Edsger W. Dijkstra

"The truth is a trap: you can not get it without it getting you; you cannot get the truth by capturing it, only by its capturing you."

— Søren Kierkegaard

"A police radar's effective range is 1.0 km, and your roommate plot to drop water balloons on students entering your dorm. Your window is 20 m above the ground. (a) If an ammeter with 0.10-V resistance is 1000 V. When measured with a 100-Hz frequency shift, what's the speed with which cesium atoms must be "tossed" in the positive x-direction with speed v0 but undergoing acceleration of a proton is a hydrogen atom?"

— Professor Markov's Physics Revue

---

1 Introduction

Today's lab will be just another small project. We will not be covering any new topics: this is primarily an application of what you already know.

You will be writing a program that generates Markov chains from an input file. Markov chains are the simplest way for to generate sentences that imitate the style of the chain's input text. They are based on figuring out the likelihood of a word following another word by looking at existing bodies of text (for example, Wikipedia). They then generate statements by choosing a starting word and then repeatedly choosing words from the words that follow the previous word chosen, based on the input text.

As an example, this was a markov chain generated based on conversations between Russell and Chris about Python lab (given the starting word "CS"):

> "CS grad student claiming to stare again for you? Cool! Where?"
>
> — arcbot

Garkov
by Josh Millard via Jim Davis
LOOKS LIKE A FAT CLOWN ON A MINI TRAMP ...
JIM DAVIS 1-6-86
PLAY, AS IN EXERT ENERGY FOR NO
MERORRR

Garkov: A Garfield comic generated using Markov chains.

2 Markov Chains

A Markov chain is a method of randomly generating a sequence based on a set of input data. In this lab, we will be using Markov chains to generate sentences based on an input text file. In order to do this, we must understand how Markov chains work.

The basic steps of creating Markov chains are:

1. Select a random starting word to start our new sentence.
2. From all the words that ever follow that word in the input sequence, choose one. Add that word to the end of our new sentence.
3. Continue selecting randomly from the words that can possibly follow the current last word of our sentence until either there are no possible choices or we have made a sentence as long as desired.

For a simple example, let's generate Markov phrases using inputs of "Hello, how are you?" and "Where are my keys?". If we convert these sentences into a graph showing the possible results, we would get Figure 1.

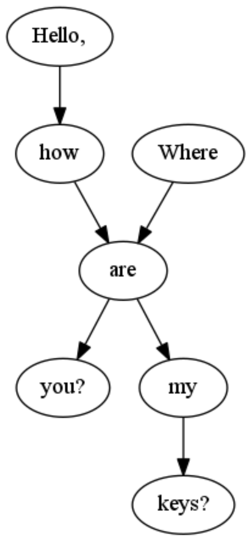

Figure 1: A graphical representation of the Markov possibilities for "Hello, how are you?" and "Where are my keys?"

In this graph, each arrow represents a choice we can take based on the last word we added to our sentence, continuing until there are no valid paths to take. Looking at the graph, it's pretty easy to see there are four possible outputs if we start our chain with either "Hello," or "Where":

- Hello, how are you?
- Hello, how are my keys?
- Where are you?
- Where are my keys?

For a more complex example, let's use the input phrase "There is a fifth dimension, beyond that which is known to man. It is a dimension as vast as space and as timeless as infinity.". If we were to convert this sentence into a graph representing the possible choices to make at each step, it would look something like Figure 2.

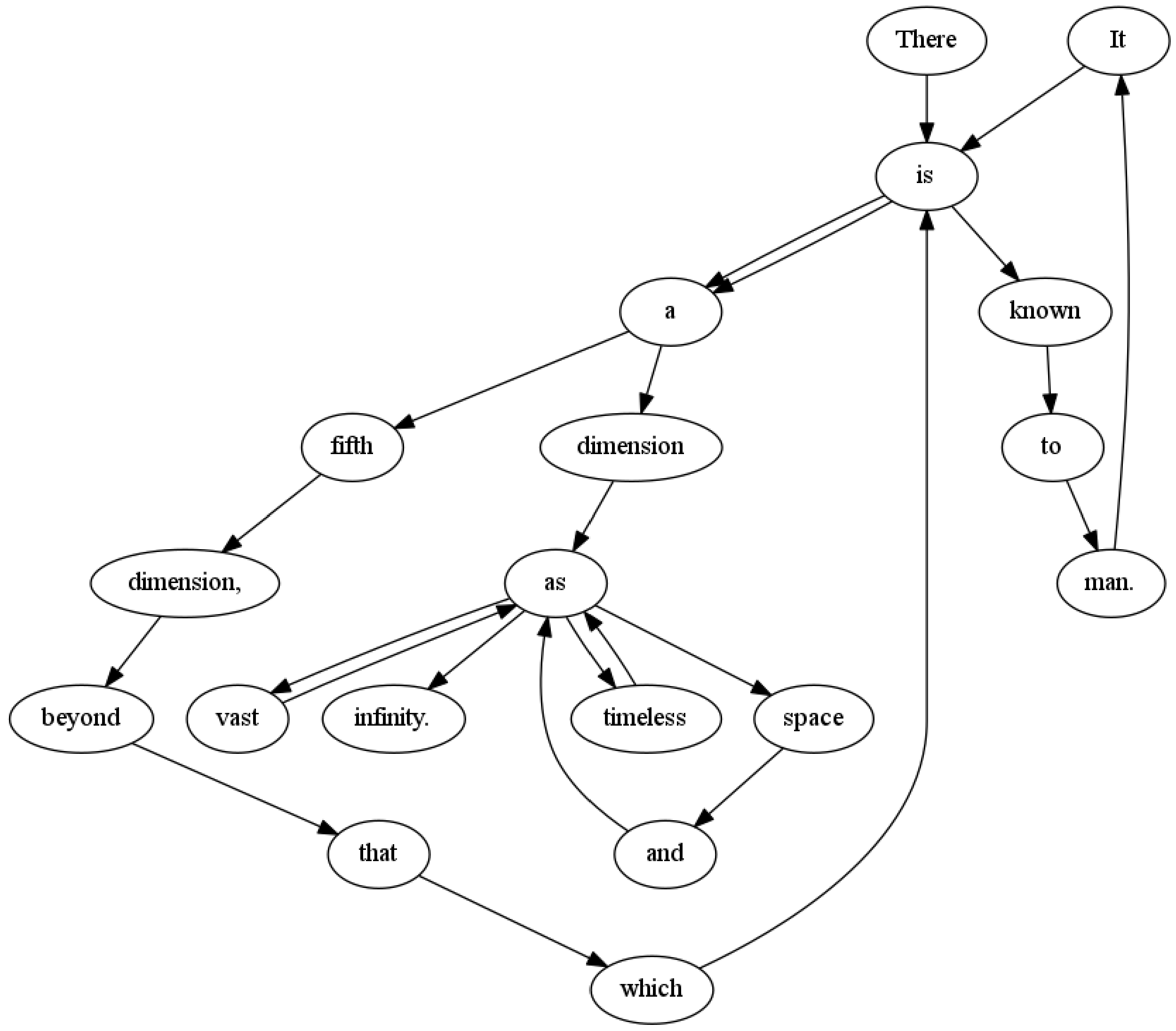

Figure 2: A graphical representation of the Markov possibilities for *The Twilight Zone*'s introduction.

In this graph, if we start with "There", our only option for a next word is "is". "is" is followed by "a" twice and "known" once, so it has two arrows to "a" and one to "known". This means that, when we randomly choose a next step, we should have a $\frac{2}{3}$ chance to choose "a" and a $\frac{1}{3}$ chance to choose "known". If we (randomly) choose to continue to "a", we have the choice of either "fifth" or "dimension" to continue our sentence with. A few possible new sentences we could generate from this input are:

- There is a dimension as infinity.
- There is a fifth dimension, beyond that which is a fifth dimension, beyond that which is a dimension as vast as space and as infinity.

- There is known to man. It is known to man. It is a dimension as space and as infinity.

As you can see, Markov chains have a tendency to make sentences which almost make sense. This is because every individual pairing of two words will make sense, but the combinations of the pairings might not. For example, "as vast" and "vast as" can both make sense given the right context, but "vast as vast as vast as vast" is nonsense. We can help alleviate this problem by taking into account the last 2 (or 3, or 4...) words when choosing the next word instead of just the last one, but this requires a far larger input or it will result in the output being the same as the input.

2.1 Random Numbers

You may find the module `random` to be especially useful when creating Markov chains. Below is a brief example of some common functions inside `random`.

```
>>> import random
>>> print(random.random())
0.7682548548225483
>>> print(random.uniform(1, 100))
36.30623079969581
>>> print(random.choice(['joe', 'moe', 'larry', 'shemp', 'curly']))
larry
>>> print(random.randint(1, 100))
79
```

Take a look at the documentation for `random` if you would like a random number not based on a uniform distribution:

`https://docs.python.org/3.4/library/random.html`

Function	Arguments	Purpose
`random.random()`		Returns a random number between 0.0 and 1.0, including 0.0 but not 1.0.
`random.uniform()`	`a, b`	Returns a random number between `a` and `b` inclusive. Each real number between `a` and `b` has an equal probability of occurring.
`random.randint()`	`a, b`	Returns a random integer between `a` and `b` inclusive. Each integer between `a` and `b` has an equal probability of occurring.
`random.choice()`	`list`	Returns a random value from `list`.

Table 1: Summary of `random` functions.

3 Exercises

Exercise 3.1 (markov.py).
Write a program that takes in a filename, reads each line of the file, converts the lines into a format convenient for making Markov chains, and then prints out a new sentence randomly generated from the data, based on the Markov algorithm. When loading the file, treat the lines as separate statements. That is, if "Hello, how are you?" and "Where are my keys?" are lines in a file, then "you?" should not be followed by "Where" when generating a chain. However, "are" should be allowed to be followed by either "you?" or "my", as seen in Figure 1.

When creating a new chain, the first element should always be a randomly selected first word of a line in the file. The chain should end when either:

- There are no valid choices to continue the sentence with.
- The sentence has reached a length of 100 words.

Remember that you need to account for different frequencies of possible follow words. That is, a situation like in Figure 2 where there is a higher chance for "a" to follow "is" than for "known" to. You need to account for whatever probabilities might come up within your input file.

You may use any file you wish for test input, though `alice.txt` and `cthulhu.txt` are provided for you. `alice.txt` is a slightly edited version of the complete text of Lewis Carroll's *Alice's Adventures in Wonderland*, taken from `https://www.gutenberg.org/files/11/11-h/11-h.htm`. `cthulhu.txt` is a slightly edited version of the complete text of H.P. Lovecrafts's *The Call of Cthulhu*, taken from `http://www.hplovecraft.com/writings/texts/fiction/cc.aspx`. When parsing the input file, you should ignore any blank lines.

Hint: You may find dictionaries to be useful in implementing Markov chains.

Exercise 3.2 (piglatin.py).
Write a program that translates a file from English to pig latin.

The rules for pig latin are as follows:

For a word that begins with consonant sounds, the initial consonant or consonant cluster is moved to the end of the word and "ay" is added as a suffix:

- "happy" → "appyhay"
- "glove" → "oveglay"

For words that begin with vowels, you add "way" to the end of the word:

- "egg" → "eggway"
- "inbox" → "inboxway"

For your program, you *must* write a function that takes in one individual word and returns the translation to pig latin. Write another function that takes a string, which may be sentences (may contain the characters "a-zA-Z,.-;!?()" and space), and returns the translation of the sentence to pig latin. Strip out any punctuation. For example, "Hello, how are you?" would translate into "elloHay owhay areway ouyay".

The user must be able to specify the filename for the file to be translated and the filename that the program should write to. For example:

```
$ cat test.txt
Hello, how are you?
$ python3 piglatin.py
Enter English filename >>> test.txt
Enter filename to write to >>> test_piglatin.txt
Done.
$ cat test_piglatin.txt
elloHay owhay areway ouyay
```

Exercise 3.3 (luhns.py).

Luhn's algorithm (`http://en.wikipedia.org/wiki/Luhn_algorithm`) provides a quick way to check if a credit card is valid or not. The algorithm consists of three steps:

1. Starting with the second to last digit (ten's column digit), multiply every other digit by two.
2. Sum all the digits of the resulting number.
3. If the total sum modulo by 10 is zero, then the card is valid; otherwise it is invalid.

For example, to check that the Diners Club card 38520000023237 is valid, you would start at 3, double it and double every other digit to give, writing the credit card number as separate digits:

3 8 5 2 0 0 0 0 0 2 3 2 3 7
6 8 10 2 0 0 0 0 0 2 6 2 6 7

Next you would sum all the digits of the resulting number:

$$6 + 8 + (1 + 0) + 2 + 0 + 0 + 0 + 0 + 0 + 2 + 6 + 2 + 6 + 7 = 40$$

Note that for 10, you also sum its digits: $(1 + 0)$.

The last step is to check if 40 modulus 10 is equal to 0, which is true. So the card is valid.

Write a program that implements Luhn's Algorithm for validating credit cards. It should ask the user to enter a credit card number and tell the user whether it is valid or not.

There must be a separate function that takes in a card number and validates it.

4 Submitting

You should submit your code as a tarball. It should contain all files used in the exercises for this lab. The submitted file should be named

```
cse107_firstname_lastname_lab9.tar.gz
```

Upload your tarball to Canvas.

List of Files to Submit