Markov Word Chains of Order 3 in Bash

CS 224

1 Introduction

Have you ever stopped to consider how *predictive text* systems work? The underlying problem is deceptively simple: given the previous N words, what are likely candidates for word N+1? There are many ways to approach this problem, which require varying levels of sophistication and training.

One approach uses what are called *Markov chains*. The idea is to use a sequence of N elements (words, in our case) to provide a likely N+1 element. Imagine we want to add a word to the sequence "I have a". We could look at that, realize we need some sort of noun phrase, and construct a noun phrase. Or, we could round up a bunch of existing sentences which begin "I have a" and pick one of their next words. This latter approach has two advantages: we don't need to understand English grammar, and we can tailor the output to a particular person or domain by considering only certain sentences. (This latter feature is why we can play predictive text games on social media.)

For this project you will construct a pair of bash scripts that can be used to generate gibberish sentences using this technique. This project will be turned in using GitLab, so you should probably take a look at Sections 5 through 7 for those requirements. In the first phase of the project (see Section 2), you'll write a script to receive an input text and break it into four-word fragments. Then, you'll write a second script (see Section 3) that will generate sentences.

You'll need a fairly significant amount of text to break into fragments if you want interesting sentences. I've included the text of *Alice in Wonderland* with this assignment, but you can try any other text you'd like. Project Gutenberg can hook you up with all sorts of works in the public domain.

Here is an example string generated from *Alice in Wonderland*:

```
in his note book hastily consider your verdict he said to the queen
```

Note that the sentence doesn't actually make sense, but it has sort of a Wonderland-y feel about it. In this case, the word `hastily` was chosen because there are two fragments that begin with "his note book":

```
his note book cackled
his note book hastily
```

and my script chose one of them at random to supply the next word.

2 Generating Word Sequences from Input Text

Your first script must be named `markov-words.sh`. It will read input from the terminal and output a list of all four-word sequences in its input.

2.1 Two-word lists

The following script splits its input into a list of pairs of successive words:

```
#!/bin/sh

PREVWORD="BEGIN"
tr -c "a-zA-Z" '\n' | grep '[a-zA-Z]' | while read WORD
do
    echo "$PREVWORD $WORD"
    PREVWORD="$WORD"
done
```

Input is passed through the pipeline where:

1. `tr` converts anything that isn't a letter into a newline
2. `grep` returns only lines with letters
3. `read` returns a word at a time into the variable `WORD`

The loop then prints the old word and the new word, then replaces the old word with the new one and repeats. Here's an example, where we just pipeline in text:

```
$ echo "The cat in the hat" | ./words.sh
BEGIN The
The cat
cat in
in the
the hat
```

2.2 Your Task

Using the sample script as a starting point, construct a bash script named `markov-words.sh` that generate a list of all four word sequences from its input:

```
$ ./markov-words.sh < alice-in-wonderland.txt > alice-words.txt
$ tail -n 4 alice-words.txt
and the happy summer
the happy summer days
happy summer days the
summer days the end
```

- The script should work directly with `stdin/stdout` (as in the sample script)
- All words should be converted to lower case (tr can help)
- The only one character words should be `a`, `i`, and `o`
- All other words should be at least two characters long and contain a vowel

It's fine if your first few lines of output include placeholders, e.g. like this:

```
^^^ ^^ ^ alice
^^ ^ alice adventures
^ alice adventures in
alice adventures in wonderland
adventures in wonderland lewis
```

3 Generating Markov Word Chains

Your second bash scripted, named `markov-sentence.sh`, uses the word list produced by the first script to generate a word chain of a given list. It should accept the name of a words file, and the target output length, as *command line arguments* and write output to the terminal, like so:

```
$ ./markov-sentence.sh alice-words.txt 15
ever be a secret kept from all the rest were quite silent and looked at
```

Your script needs to verify that its arguments make sense before attempting to run.

3.1 The Algorithm

`markov-sentence.sh` needs to implement the following algorithm:

1. Select three starting words
2. Find all fragments which begin with those three words
3. Randomly select one of their fourth words
4. Repeat the process, using starting words 2 and 3, along with the fourth

We'll look at each step in turn.

3.2 Step 0: Checking Command-Line Arguments

Programs should be sure their input is sensible before they act on it, and scripts are no exception. Have your script verify that:

- The first argument is a file which actually exists
- The specified file is readable
- The specified number is greater than 0

`test` (aka `[`) is likely to be your friend for this.

3.3 Step 1: Pick Starting Words

We want to try and make sure we can actually generate output, so it makes sense to pick the starting words from the words file. So, pick a random line in the file (all of the lines should have four words) and use the last three words in the line as your start point.

3.3.1 Getting a Random Line

There is a semi-standard program named `shuf`, whose job it is to randomly reorder the lines of its input and print that out. Assuming a working `shuf`, you can get a random line from a file by

1. Shuffling the file's contents
2. Taking the first line of output as your random line

Unfortunately, you may not have `shuf` on the system you're using, so I have provided you with a small C program (shuffle.c, in Canvas) which does the same thing. If you compile the program with `gcc -o shuffle shuffle.c` you will end up with an executable named `shuffle`.

How do you have your script call the right program? Include the following near the top of your script:

```
SHUF=$(command -v shuf)
if [ ! -x "$SHUF" ]; then
    if [ -x "./shuffle" ]; then
        SHUF="./shuffle"
    else
        echo "No shuffle program!" 1>& 2
        exit -1
    fi
fi
```

After it runs, the variable `SHUF` will be populated with a working shuffle program (either `shuf` or `./shuffle`) so you can just call it.

It get a random line from a file, you can do something like this:

```
LINE=$($SHUF < $INPUT_FILE | head -n 1)
```

or a random line beginning with three specific words.

```
LINE=$(grep "^the rabbit asked" $INPUT_FILE | $SHUF | head -n 1)
```

3.4 Step 2: Find Fragments Beginning with Selected Words

Once you have your starting words (the last three words on a line) find all of the lines in the input file that *begin* with those words. `grep` is likely to be your friend here.

3.5 Step 3: Pick a Random Fourth Word

Once you have a list of candidate lines, pick one of them at random. That line's fourth word becomes your new word. Print it out.

3.6 Step 4: Rinse and Repeat

Repeat the above process until you have found the correct number of words. The initial seed words (the first three, chosen from a random line) do not count toward the total number of words selected. The first word picked using the 'find all words beginning with these three words' mechanism is the first word for the purposes of output counting.

You might want to take a look at the `seq` program / command.

3.6.1 Special Case: Hitting End of File

There's an outside possibility that your script will end up picking the last couple of lines of the words file as starting points. In that case, you're going to hit the end of the file and then not find any candidates for continuing. If that appears to have happened, We'll run your program again so it can (hopefully) find a better starting point.

3.7 Sample Run

This program involves randomness, so you might see different results, but that's ok. So long as your program behaves properly you should be good.

Let's imagine our script was called as follows:

```
$ ./markov-sentence.sh alice-words.txt 5
```

After checking the input, we find a random line to start us off:

```
in his note book
```

The last three words give us our seed, so find all lines that start with them:

```
his note book cackled
his note book hastily
```

Say we randomly pick "cackled" to be the next word. It becomes the first word in our output, and we find all line books beginning with "note book cackled":

```
note book cackled out
```

There's only one candidate but that's ok. Pick word four ("out"), output it, and get the fragments beginning "book cackled out":

```
book cackled out silence
```

Pick "silence", repeat:

```
cackled out silence and
```

Pick "and", repeat:

```
out silence and read
```

At this point we've gone through five passes, as requested, so the output is

```
cackled out silence and read
```

Two things are of note here:

- You don't actually need to save the output you're generating – just print it
- It's entirely possible to hit a sequence where you only have one choice

Generally speaking, the larger your *corpus* of input data, the more unique sentences you'll be able to create.

Here is a longer example run:

```
$ ./markov-sentence.sh alice-words.txt 400 | fold -w 80 -s
moment splash she was up to her chin in salt water her first idea was that she
had put the lizard in head downwards and the poor little thing was snorting
like a steam engine when she caught it and kept doubling itself up and
straightening itself out again so that altogether for the first minute or two
it was as much as she could if you do know what a mock turtle is it the gryphon
lifted up both its paws in surprise what never heard of uglifying it exclaimed
you know what it means i know what it means well enough when i find a thing
said the queen turning purple i won said alice off with her head about once in
a large ring with the mouse in the middle of her favourite word moral and the
```

```
arm that was linked into hers began to tremble alice looked up and there was
silence for some minutes alice thought to herself i the cheshire cat sitting on
a bough of a tree did you say what a pity the rabbit asked no i didn said alice
i don even know what they re like i believe so alice replied thoughtfully they
have their tails in their mouths and they re their arguments to her though as
they all spoke at once she found it very hard indeed to make out exactly what
they said the executioner argument was that anything that had a head could be
beheaded and that you weren to talk nonsense the queen argument was that if
something wasn done about it in less than no time she have everybody executed
all round it was this last remark it a vegetable it doesn look like one but it
is i quite agree with you said the hatter you must remember remarked the king
or i have you executed on the spot this did not seem to encourage the witness
at all he kept shifting from one foot to the other looking uneasily at the
queen and in his confusion he bit a large piece out of his pocket and was
looking at the place where it had been it suddenly appeared again the bye what
became of the baby said the cat alice thought she might as well go back and see
how the game was in such confusion that she never knew whether it
```

4 Additional Files

I have provided both the shuffle program `shuffle.c` and a copy of the text of Alice in Wonderland on Canvas. Please note that while your word list should match the example in Section 2, the output of `markov-sentence.sh` will almost certainly be different.

5 How To Submit

You will be submitting this project using GitLab. Begin by creating a project named `CS224-Markov` and getting it checked out and ready to use. A README is not required, but it's a good habit to get into. Do your development work as usual, checking things into git (and pushing to GitLab) as appropriate. Do not include Alice in Wonderland or any of your output files (e.g. alice-words.txt) in your repository. You may include `shuffle.c`, but do not include the `shuffle` executable. Minimally, I will expect one commit for `markov-words.sh` and one for `markov-sentence.sh`, with good commit messages. Once you are satisfied with your code, add me (paul.bonamy) as a Reporter and leave a note in Canvas that your assignment is ready to grade.

6 Good Times to `commit`

You're only required to make two commits, but it's a good idea to have more than that. Generally, you should commit changes after you have gotten something meaningful accomplished. For example, you might start the first script (markov-words) by copy-pasting in the two-word example. This is a good time for a commit, as it establishes a starting point. You can then add commits when, for example, you figure out how to convert the text to lowercase, or filter out illegal words, or get to four words per line (all of which need to happen, but could happen in any order). The goal of commits is to preserve the work you've done, so anything you don't want to break is a good candidate for a commit.

Similarly, the second script can and should be approached in (committable) pieces:

- Argument checking
- Getting a random line of text
- Finding lines based on your seed

- Picking random lines from same
- etc

A commit doesn't have to wait until something is *done*, it can happen any time you've *made progress*. It's like good save points in video games: you made it through something tricky (or are about to do something hard) so take a moment to save.

7 Grading

This assignment is worth 35 points, with the following breakdown. I reserve the right to deduct for blatantly bad style, so use reasonable variable names, etc.

	GitLab
2	Commit for `markov-words.sh`
2	Commit for `markov-sentence.sh`
6	Reasonable commit messages

	`markov-words.sh`
1	Script uses stdin / stdout (no file names are used)
1	Words are converted to lower case
1	Only single-character words are a, i, o
1	Other words are 2+ characters and contain a vowel
6	Script produces a list of four-word sequences

	`markov-sentence.sh`
1	Script uses command-line arguments
1	Script checks that file exists / is readable
1	Script checks that number is greater than 0
1	File name is not hard-coded
1	Script runs in less than a few seconds
2	Last three words of a random line used as a seed
8	Output constructed properly