Project 2: Markov Babbler

Due: 27 September 2023

v20230918-1245

In this project you'll write a text generator that is analogous to the core piece of ChatGPT (and other generative AI systems): a Markov babbler.

The exact details of such generators will be covered in class over the next few days, but the basic idea is that you have to answer the question, "based on what came before, what is the most likely next word that should be generated?"—and then emit that word. To do this, your program will need to train itself on an established body of data—known as a *corpus*[1]—that contains a lot of words and learn some statistics about the sorts of words that appear in sequence with each other. (Relative to ChatGPT: this is purely a language model, and not a particularly sophisticated one, so it's not going to be writing essays with meaningful content. But technically speaking, it will be a generative AI system!)

Objectives

In the course of this project, the successful student will:

- implement a Markov model based on empirically-observed data,
- manage multiple open files to read data, distributed throughout several directories in the filesystem,
- use maps/dictionaries to maintain frequency information of words in different contexts, and
- apply standard techniques for probabilistic computation in the presence of very small numbers and absent data.

The data sets

I have access to the North American News Corpus (NANC) (Graff 1995) and have uploaded a slightly processed version of it to `/home/shared/nlp`. There are also some other files in that directory that you may or may

[1] Plural *corpora*, which is pronounced COR-prr-ruh, because Latin.

not find useful. For purposes of this project you will be working with the `/home/shared/nlp/nanc-txt/` directory.[2] Each file generally represents (a selection of) the articles from a particular news publication for one day. Each article ends with a separator "`=====`".

Inside the directory is a file named `files` with a full listing of the files in the various subdirectories. It's relative to the directory itself, and with the name of the directory and basic string concatenation you should be able to put together filename that you can open with an `ifstream` or a `Scanner` or whatever. (Even if you *do* know how to list directories at runtime, you should still use the file index—it makes it a little easier to build and control your test cases. If you are thinking about reaching for a `DIR*` and `dirent`, you are barking up the completely wrong tree.)

I/O spec

Just so we're on the same page (and I can automate some things) I'm requiring you to specify the running parameters as *command line parameters*,[3] in the following order:

programname *what degree words dir index*

where:

- *what* is one of "`dump`", "`gen`", "`eval`", or a decimal number like 0.4
- *degree* is 0, 1, or 2
- *words* is a maximum number of words to produce, such as 50
- *dir* is a directory with training data files in it
- *index* is a file with filenames relative to that directory

Even in early phases like the prep work, where some of those options are irrelevant, you can *ignore* the command line option but all of them should still be in the same positions on the line (and if a user doesn't provide all of them you can exit with an error message or do some other graceful thing).

The program will then read one line (or, later, more) from standard input, and print its primary results to standard output. If you also have debugging

[2] You might also have fun playing around with the `gutenberg` files to make your babbler speak in the style of Jane Austen, William Shakespeare, etc, but that's not required; and it doesn't currently have the same files-and-index layout as the NANC directory.

[3] If you're not sure how, look it up or see me. In C++ this involves using `argc` and `argv`; in Java, the `args` array; and in other languages there are other useful incantations.

statements or user prompting, print that to standard error (`cerr` or the equivalent).

Prep work

Unlike the last project, I'm not giving you any code to start from. Your initial work involves getting the outer shell of the program running: getting the directory name and index file from the command line, reading the filenames it lists, and looping through to open and process each of the files.

To demonstrate that you've done this, your checkpoint program will read and store the first whitespace-separated "word" in each referenced file, then after the user presses Enter (i.e. your program reads a line from standard input), print (to standard output) those words.

(Suggestion: make a local version of `files` that has a much smaller number of filenames in it, like two or three. No need to work with all 6,486 files in the corpus right off the bat!)

Thus if you made a index that had just the first six lines of `files` and ran your prep work with that file, it would wait for the user to hit Enter and then print

```
Anyone
WASHINGTON
WASHINGTON
WASHINGTON
LOS
RIVERSIDE,
```

(a lot of these articles start with a location!)

This work is due next Wednesday **at 8pm**. You'll have a chance to ask questions about it in class on Wednesday to clear up any last-minute issues, but you really don't want to wait until then to start it. When you're ready to hand it in, use the `handin` script as described at the end of this document.

A little more about the task

A Markov babbler emits text at random based on a probability distribution learned from a corpus of text. Output from high-order Markov babblers can appear quite uncanny, as if they *almost* mean something.

Any generative model is fundamentally based on the idea that you have a probability model

$$p(\text{next thing}|\text{previous stuff}, \text{other context})$$

For a text-based model in particular, we can refine that: the probability of a particular word, say word #7, is based on what came before:

$$p(w_7 = \text{“the”}|\text{other words in the sentence}, \text{other context})$$

The core idea in a Markov model is the Markov assumption, that the only relevant context for deciding what word someone says next is (some number of) the most recent previous words; not the words before that, and not the position in the sentence. For instance, in a first-order Markov model, the probability is based on the one preceding word:

$$p(w_i = \text{“the”}|w_{i-1})$$

This assumption makes it quite easy to build a generative model, although it does sometimes give rise to some implausible or even ungrammatical sentences, e.g.

> The boy rapidly walk to her own a car.

Each small section of that line is a reasonably probable combination, but the long-distance dependencies make it problematic. Fortunately, that's a problem with the model, and one you won't need to "fix" in your program. (In fact, the weirdness of the output is a large part of what gives Markov babblers their charm!)

Design work

Once you've got the shell of the program running (or perhaps while you are working on that, but the design work depends on course content we won't cover until next week), you can start thinking about the algorithm and data structure design.

1. Devise an extremely short and simple data set that can serve to fuel test cases. It should have just a small handful of files, each with just a few words; their collective contents should be able to test the different aspects of the empirical data collection system (in particular, some words and word combinations should be repeated, and some not).

2. What built-in library data structures will you use to store the frequency information you'll need to compute the probabilities? Express the types of these data structures in the programming language you intend to use (and verify that its library includes those types!). Give descriptive names to them so you'll remember what they're for; and write out what the contents of those data structures would be given the data example you wrote out in the previous item.

3. The end goal will be to evaluate probabilities and generate text based on probabilities. Write pseudocode for how you will take a probability distribution (i.e. likelihoods for each possible "next word") and use it to choose what word to generate next.

4. How will you handle it when the prompt contains words that don't exist?

5. The full NANC corpus has about a billion words of text; even dealing with a substantially smaller portion of it will mean handling a *lot* of data. What are the likely speed bottlenecks in the project? How can you use data structures to manage them appropriately?

Final version

A full-credit final version will be a complete, non-buggy, working implementation of a Markov model that can train on a lot of data and either generate or evaluate text based on that model, TOGETHER WITH convincing proof that it is correct. The program should be able to read from the provided training corpus and then behave according to the command-line parameters.

The data from the corpus is currently stored as regular running text; your program should tokenise it as follows. In addition to shifting all letters to lowercase, it should pull punctuation that starts or ends a "word" into its own separate token (while leaving mid-word punctuation like hyphens and apostrophes alone), and while multicharacter punctuation can mostly

remain together, it should pull any of these six characters—`,;:.?!`—into their own token, and a punctuation group that contains sentence-ending punctuation (`.?!`) should be followed by the special token `STOP` indicating a (probable) sentence boundary. So the input

```
``No; I think,'' said Mr. Addle-Ross, ``it isn't!''
```

should tokenise into

```
`` no ; i think , '' said mr . STOP addle-ross , `` it
isn't !  ''  STOP
```

The presence of the `STOP` is just a quick-and-dirty best-guess way to identify ends of sentences; it's sometimes incorrect (as above) but that's outside the scope of the project. When generating, a generated `STOP` shouldn't be printed but instead indicates the end of the utterance (and time to await further user input).

The full data from all files in the index file should be read and stored; assume an implied `STOP` precedes the contents of each file.

The user input is tokenised using the exact same rules as for the training.

The command-line argument indicating what mode the program is running in is interpreted as follows:

dump tells it to print its trained data—essentially, debugging output, and you probably don't want to use this option when trained on the full data set.

gen tells it to generate directly from the language model; for each new word, it will consider all possible next words and choose one with likelihood corresponding to the probability assigned by the model.

eval tells it to take the user's input as a sequence of words and reports the probability the model would assign to that sentence. These numbers will be very small and should probably be reported in some version of scientific notation (with large negative exponents)

0.4 or some other number tells it to generate from the language model using that number as the "temperature"—lower numbers are more likely to just generate the single most likely word, higher numbers are more "creative".

Whether generating or evaluating, the program should not terminate until the user input does; each line of user input should be taken as a "prompt" (when generating) or as a sentence to evaluate (when evaluating). An empty line can still be considered a prompt, with no words in it, and should still generate a response. (If interacting at the command line, Ctrl-D indicates that user input is done.)

The parameter dictating the degree of the Markov model is either 0, 1, or 2: this indicates how many previous words will be taken into account by the model. (These are also known as "unigram", "bigram", and "trigram" models respectively.)

When a user prompt is less than two tokens long, or includes words that are not known to the model (or perhaps in contexts not known to the model), it should still behave gracefully. Similarly, when evaluating sentences, it should not crash or print zero when presented with previously unseen words and word combinations.

Handing in

Hand it in as `proj2` using the handin script. The prep work is due 8pm on the 13th, and the final handin is due 8pm on the 27th.

Reference

D. Graff, *North American News Text Corpus LDC95T21.* Web Download. Philadelphia: Linguistic Data Consortium, 1995.

Rubric

RUBRIC

General (50)
15 Prep work
10 Design
10 Documentation
15 Followup questions
File and text processing (27)
10 Opens and records at least one token from each file in given directory that is listed in given index (i.e. the prep work)
5 Segments all tokens in every file (at least on whitespace)
5 Converts words to all lowercase with no start/end punctuation
5 Retains punctuation as separate tokens
2 ... with ,.;:!? further separated out
Counting and probability (35)
10 Counts unigram frequency of each token
5 Counts bigrams and their probs (Order-1 Markov model)
5 Uses prompt when generating output
5 Counts implied STOP token before files and after sentence-ending punctuation, and uses it when generating to stop output
5 Counts trigrams and their probs (Order-2 Markov model)
5 Uses appropriate fallback if prompt's words are omitted or unknown, when generating and evaluating
Generation and evaluation (30)
5 Generates text based on provided files with any randomness
5 Generates text directly from probabilities in language model
10 Evaluates sentence in input and prints its probability
10 Generates text using language model and temperature
Program flexibility (8)
2 Behaves to dump/gen/eval/temperature according to CL param
2 Varies order of MM according to CL param
2 Varies number of words generated according to CL param
2 Responds to multiple prompts (either gen or eval) until user ends program