COMPSCI 260, Introduction to Computational Genomics **11 November 2022**

Problem Set 6

Due: 5pm on ***Wednesday 23*** *November 2022*

Problem 1: Don't worry yourself sick (10 points)

List of files to submit:

1. `README.problem1.[txt/pdf]`

A friend of yours, Murray Worry, is something of a hypochondriac. One night, while watching the news, Murray learns about a rare disease estimated to occur in one of every 40,000 people. Despite the odds, he insists you accompany him to get a proper medical test in order to rule it out. After an excruciating wait at the clinic, you and Murray are finally welcomed back to see Dr. Mel Practus. In an effort to reassure him, Dr. Mel indulges Murray and orders the test because she knows it to be 99.7% accurate (specifically, if a patient has the condition, there's a 99.7% chance that the test detects it, and if the patient doesn't have the condition, there's a 99.7% chance that the test does not detect it). After a quick swab from the inside of Murray's cheek, Dr. Mel excuses herself from the room to carry out the diagnostic test. Moments later she returns and informs you both of the bad news: the test has come back positive.

a) Might there be some quantitative way you can comfort your friend? Think carefully about how likely it is that Murray actually has the disease, given that he tested positive.

b) How do you explain your answer to part (a) in light of the fact that the test is 99.7% accurate? Among all instances of a positive result from a diagnostic test, the fraction of those coming from people who do not actually have the disease is called the false discovery rate; what would have to be true about this diagnostic test in order to observe a more reasonable false discovery rate?

c) What if it turns out that this rare disease develops as the result of very specific environmental factors? Does your conclusion from part (a) still hold? Why or why not?

Problem 2: Writing short stories like Joseph Conrad (25 points)

List of files to submit:

1. `README.problem2.[txt/pdf]`
2. `language.py`
3. `heart.of.darkness.mm.[1-4].txt (one file for each of the four Markov models)`
4. `paradise.lost.mm.[1-4].txt (one file for each of the four Markov models)`

Plan

Markov models are popular tools in computational biology, because they are useful any place you encounter sequential data that has spatial or temporal dependence within the data (including other applications like speech recognition, handwriting recognition, information retrieval, financial analysis, and data compression).

In this problem, we will explore a common application of a Markov model: generating artificial sequences of characters that capture the style of the sequences of characters on which the model was trained. Put another way, you will design and implement a Markov model that can generate stylized text, where the style is determined by the input text used to train the model. Markov models of this kind are used often with genomic data, but it will be more beneficial to your intuition to work with English text. The result of your work will be a program that writes an English short story for you in the style of Joseph Conrad, or a poem in the style of Milton; it's not as hard as it might seem at first.

Characters in English text are not spatially independent: If the previous five characters in a string of English text are `arith`, then the next letter is almost surely an `m`. A k^{th} order Markov model predicts that each character in a text occurs with a probability that *does not* depend on the location it occurs in the text, but *does* depend on the previous k characters. These probability values are *parameters* of the model, and we can perform parameter estimation to learn these probability values from observed data (this is what we mean by 'training' the model).

In short, we train a Markov model to generate text in a certain style by estimating the parameters of the model on the basis of input text with that style. We do this training, or parameter estimation, using approaches like maximum likelihood (empirically observed frequencies, which are based on occurrence counts) or maximum *a posteriori* (based instead on occurrence counts plus pseudocounts).[1] In particular, under the maximum likelihood principle you can estimate the parameters of a Markov model of order k using empirical frequencies: counting up how often each character occurs following each sequence of k characters. For example, if the text has 100 occurrences of `th`, and 50 of those occurrences occur within `the`, 25 within `thi`, 20 within `tha`, and 5 within `tho`, a second order Markov model would predict that the next character following a `th` is `e` with probability 1/2, `i` with probability 1/4, `a` with probability 1/5, and `o` with probability 1/20.

Note that if we train a Markov model on English texts, we will learn different parameters than if we train it on texts from different languages. Amazingly, even a simple second order Markov model is sufficient to reveal the language used to train the model; consider these examples:

- jou mouplas de monnernaissains deme us vreh bre tu de toucheur dimmere lles mar elame re a ver il douvents so
- bet ereiner sommeit sinach gan turhatter aum wie best alliender taussichelle laufurcht er bleindeseit uber konn
- rama de lla el guia imo sus condias su e uncondadado dea mare to buerbalia nue y herarsin de se sus suparoceda
- et ligercum siteci libemus acerelen te vicaescerum pe non sum minus uterne ut in arion popomin se inqueneque ira

These are artificial texts generated by second order Markov models trained from texts in French, German, Spanish, and Latin, respectively. So while these outputs are completely non-sensical, even second order models are sufficient to capture something of the flavor of the original texts.

In what follows, we walk you through the steps for training a Markov model on an input text and then generating artificial text from the trained model. We provide you with a starting template in the Python file `language.py`. At the highest level, this code specifies the name of a file and the order k of the Markov model, and then calls `generate_mm_text`. In turn, `generate_mm_text` will load the contents of the specified file as a training text, and will make use of two functions you will be asked to implement later: `collect_counts` and `generate_next_character`.

[1] Here you should definitely use maximum likelihood—the fact that not every combination of letters occurs in the English language will make our dictionary smaller, so we won't use pseudocounts to compensate for letter combinations that never occur in the training text. (As an aside, there are ways to factor in pseudocounts without actually making the dictionary bigger, but we'll leave that as a thought exercise: Can you think how?)

Develop

Step 1: You need to be able to count up occurrences of all the various k-tuples in the training text

a) To ease you into things before you write the `collect_counts` function, first write a simpler version of that function called `build_dict`, which will consider each consecutive k-character substring (a.k.a. k-tuple) that appears in the text and insert it into a dictionary with the k-tuple as the key and the total number of occurrences of the k-tuple as the value. For example, if $k = 2$ and the input string is `agggcagcgggcg`, then the program should insert into a dictionary a total of five keys (namely `ag`, `gg`, `gc`, `ca`, and `cg`) and each key should have a value equal to the number of occurrences of that key (namely 2, 4, 3, 1, and 1, respectively).

Note: For reasons that will become apparent in the next step, your function should *not* count the very last k-tuple in the text (in this example input string, that would be `cg`) because no character follows it. We have provided a function `display_dict` that prints out the number of distinct keys as well as the number of times each key appears in the text. If you've done everything correctly, the result of displaying the dictionary built from our example string should look something like this:

key	count
ag	2
ca	1
cg	1
gc	3
gg	4

Note also: Your code should not hard-code any limiting assumptions about the string that is provided as input, apart from the fact that it will contain visible characters of the kind Python normally handles. So, for example, you shouldn't assume that you will only ever see lowercase letters and spaces, even though that's what we will be using later in training. It also means you should treat lower case letters and upper case letters as different characters (e.g., `a` is different from `A`).

Step 2: You need to collect the counts required to train the model (estimate the transition probabilities)

Given a k-character key, you will want to have at hand the frequency counts of each of the characters that follow it in the training text (we nickname these as 'followers'). This operation is at the crux of the matter, since you will need the counts to estimate the transition probabilities that are used to generate random characters in accordance with the Markov model.

b) Write the code for the `collect_counts()` function, which creates a dictionary that maps a k-character key to both its frequency count and information about its followers. You will find this code repeats the k-character counting code you wrote in `build_dict`, so you should use its implementation as a starting point.[2]

You should also modify `display_dict` so that it displays this additional information. For example, if you've done everything correctly, the result of displaying the dictionary built from our example string after `collect_counts` should look something like this:

key	count	follower counts
ag	2	c:1 g:1
ca	1	g:1
cg	1	g:1
gc	3	a:1 g:2
gg	4	c:2 g:2

[2]We have chosen to separate the k-character counting and follower counting to explain each as distinct concepts. Specifically, previously the value for each key was just an integer number of occurrences, but now you'll also need to keep track of how many of those occurrences are associated with each character that follows an occurrence in the training text. The code skeleton shows the specific structure of the dictionary we'll use to keep track of things.

Step 3: You need to generate artificial text from the trained model

c) Next, you will need to write a function called `generate_next_character` that takes as input a string of k characters and returns a pseudo-random next character according to the Markov model. To accomplish this, your function will use the dictionary from `collect_counts` in the previous step. For example, with the dictionary compiled from the example input string above, `generate_next_character("gg")` should return either a `"c"` or a `"g"`, each being returned with probability 1/2.

Once you have this last function written, the `generate_mm_text` function we provided should be ready to generate a sequence of characters, starting with the first k characters from the original text, and then repeatedly generating successive pseudo-random characters. After generating a character, `generate_mm_text` shifts over one character position and generates another, always using the last k characters generated to determine the probabilities for the next character. For example, if the order of the Markov model is 2, and if it calls `generate_next_character("gg")` and gets back a `"c"`, then it would next call `generate_next_character("gc")`, which would be expected to return either a `"g"` (with probability 2/3) or an `"a"` (with probability 1/3).

The `generate_mm_text` function will continue this process until it has generated a total of M characters.

Apply

d) Once you have all this working, train first order, second order, third order, and fourth order Markov models to generate increasingly complex versions of a short story in the style of one of the masters of the short story: Joseph Conrad. You should train your various Markov models on the short story "Heart of Darkness", whose text is found in the file `heart.of.darkness.txt`. You should first examine and then use the `tidy_text.py` script we provide to clean up the text before using it as input to your program—this will result in an input text file (`tidy.heart.of.darkness.txt`) with only 27 possible symbols (the 26 lowercase letters and space). Finally, you should set $M = 1000$ for each of your four versions. Save your four versions in four different text files and submit them with the appropriate order of the model (e.g. `heart.of.darkness.mm.1.txt` for a first order Markov model).

e) Repeat this process to train first order, second order, third order, and fourth order Markov models to generate increasingly complex versions of a poem in the style of poetic legend John Milton. You should train your various Markov models on the epic poem "Paradise Lost", whose text is found in the file `paradise.lost.txt`. You should again use the `tidy_text.py` script we provide to clean up the text before using it as input to your program. Once again, save your four versions in four different text files and submit them with the appropriate order of the model (e.g. `paradise.lost.mm.1.txt` for a first order Markov model).

Reflect

f) *Look closely at the 8 artificial texts you generated in the previous subproblem. How noticeably do the Conrad artificial texts improve as the order increases from 1 to 4, if at all? How noticeably do the Milton artificial texts improve as the order increases from 1 to 4, if at all? At a fixed order, compare the Conrad and Milton artificial texts: how easy is it to tell which artificial text was trained on which source text? How does this change as the order increases from 1 to 4?*

g) You may notice that even a fourth order Markov model is insufficient to generate text that would pass muster in your writing seminar. To make the output sufficiently realistic requires that we move from a Markov model with states as letters to a Markov model with states as words. That is, we would consider the probability of generating a word conditional on the k words that precede it. *Comment 1) on why a word-based model is likely to produce more realistic output, and 2) what new computational issues would arise to make this a bit more challenging than using a letter-based Markov model.*

h) Let's say that `tidy_text.py`, instead of stripping out all of the punctuation, generated a training text that retained every occurrence of a period. Thus, after cleaning, the training text would now be comprised of 28 possible symbols (the same 27 as before, plus the period symbol). *Comment on how the output of your Markov models would change, and how periods would likely be distributed in the output.*

Note, and seriously challenging extra challenge: This same basic strategy would also work for music; if you feel like undertaking an interesting project in your spare time, you could write a program that is trained to improvise Charlie Parker jazz, Bach fugues, or Beethoven sonatas, depending on what you train it with. If you're serious about doing this, talk to Alex because he has some modeling ideas.

Problem 3: Generating an artificial genomic sequence (25 points)

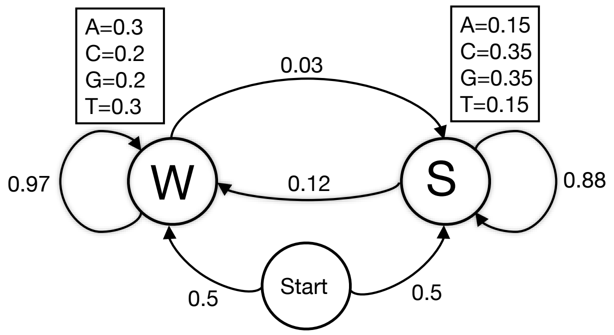
List of files to submit:

1. `README.problem3.[txt/pdf]`
2. `generate_HMM_sequence.py`
3. `analyze_sequence.py`
4. `nucleotide_sequence.txt`
5. `FLAG.txt`

Plan

Note: All calculations in Problem 3 require you to show your work to receive full credit. For example, if an answer requires you to compute a product or a ratio, please write down the numbers being multiplied or divided.

Now we will transition from English literature back to the topic of computational genomics, and we will also transition from using a Markov model of a sequence to using a hidden Markov model (HMM) of a sequence. In this HMM, let us imagine that we have hidden states W and S, which correspond to regions of the genome that are more AT-rich or more GC-rich, respectively. At each nucleotide position of the double helix, an AT pair will have two hydrogen bonds while a GC pair will have three, so the bond between strands is a bit weaker (W) in AT-rich regions and a bit stronger (S) in GC-rich regions. We will use an HMM to generate an artificial genomic sequence, stochastically alternating between these two kinds of regions (see the state transition diagram below, with the transition probabilities labeling the edges, and the nucleotide emission probabilities in the tables associated with the two states).



a) Enumerate all the possible paths of length 4 and compute the probability of each, assuming the initial state is equally likely to be W or S, as indicated in the diagram.

b) Assuming the initial state is equally likely to be W or S, compute the probability of going through the sequence of states $WWSS$ and observing the sequence $TCGA$. Is this probability the same as the probability of the path being $WWSS$, given that you observed the sequence $TCGA$? Why or why not?

c) Which of the two probabilities $P(\Pi = WWSS|X = TCGA)$ and $P(\Pi = WSSW|X = TCGA)$ is larger? How do you interpret the relationship between these two probabilities in terms of the transition and the emission probabilities? (i.e., How do the transition and emission probabilities account for this relationship?)

d) Given the length 4 observation $TCGA$, what path or paths (out of all the possible paths of length 4) do you think is most likely? (You don't have to do any lengthy computations to arrive at an answer; in fact, if you're developing the right intuitions, you should be able to explain how you can arrive at an answer without doing lengthy computations.)

e) Let's imagine we are now using our HMM to generate a very long genome sequence (along with the corresponding sequence of hidden states). Separately for each of the two states, provide a guess about how long the HMM will remain in that state on average before it transitions to the other state. Explain the intuition behind your guesses.

Develop

f) Complete the implementation of the `generate_HMM_sequence` function in `generate_HMM_sequence.py`. As with the code you wrote in Problem 2, this function will be generating a random sequence of characters, but now you will need to account for the random sequence of hidden states (which evolve with Markovian transition probabilities), and then emit the corresponding output symbols according to state-dependent emission probabilities. Your code should be able to generate both the state sequence and the symbol sequence for an arbitrary HMM, as specified within a file (we have already provided the code for loading such a file and processing it on the assumption it is properly formatted (in the real world, we'd want to validate its format); you can see an example of such a file in `HMM.parameters.txt`). Your code can assume that the states are indicated with single characters, and that the symbols are also indicated with single characters, but it shouldn't assume how many states or how many symbols there are.

In generating a state and symbol sequence, your model should repeatedly either transition to a new state or remain in the same state according to the specified transition probabilities. When the next state is determined, the model should then emit a symbol according to the emission probabilities of the state it's in at that position.

Once you have written the `generate_HMM_sequence` function, move to the file `analyze_sequence.py` which will be importing it. Within `analyze_sequence.py`, use the imported `generate_HMM_sequence` function to generate a sequence of length 100,000 using the HMM whose state diagram is shown above (its parameters should be the ones in `HMM.parameters.txt`).

Save the sequence of hidden states at each position, along with the sequence of generated nucleotide symbols, in a text file named `nucleotide_sequence.txt`. Your output should be formatted to look like this:

```
SSSSWWWWWWS...
AGTCAGGTTCA...
```

Note: This will result in two very long lines that are not very friendly for human interpretation, but please leave them like this in the file `nucleotide_sequence.txt` as this will be the expected format for the autograder.

g) Within `analyze_sequence.py`, write a function called `compute_state_frequencies` that will take an HMM-generated sequence of states and symbols as input, along with a particular symbol subsequence of interest, and will calculate and return the counts of the various hidden state sequences that prevailed whenever that subsequence of interest was observed in the output.

You will use this function in the next subproblem.

Apply

h) Apply the `compute_state_frequencies` function to the sequences of 100,000 hidden states and nucleotide symbols generated by `generate_HMM_sequence` along with the symbol subsequence $TCGA$. Use the results to **report** the frequencies of all the hidden state sequences that prevailed when the observed nucleotide sequence was $TCGA$. For the specific hidden state sequences $WWSS$ and $WSSW$, are your empirical results consistent with what you answered above in subproblem (c)? Why or why not?

i) Complete the function `compute_average_length` in `analyze_sequence.py` and use it to compute how long the system remains in each of the two states on average for the sequence of length 100,000 you generated. Have your code **report** the results returned by this function.

Reflect

j) Look at the full table of results your code reported above in subproblem (h). Empirically, which hidden state sequence prevailed most frequently when the observed nucleotide sequence was $TCGA$? *How does this comport with what you answered above in subproblem (d)? If there are any differences, explain why they arose.*

k) Look at the two average lengths your code reported above in subproblem (i). *How do the values compare with what you answered above in subproblem (e)? If there are any differences, explain why they may have arisen.*