COMPSCI 260, Introduction to Computational Genomics **8 November 2019**

Problem Set 6

Due: 5pm on 22 November 2019

Problem 1: Writing short stories like Joseph Conrad (40 points)

List of files to submit

1. `README.problem1.[txt/pdf]`
2. `language.py`
3. `hmm.heart.of.darkness.[1-4].txt` (one file for each of the four Markov models)
4. `hmm.paradise.lost.[1-4].txt` (one file for each of the four Markov models)

Plan

In addition to their usefulness in computational biology, Markov models are popular tools any place you encounter sequential data with spatial or temporal dependence: speech recognition, handwriting recognition, information retrieval, financial analysis, and data compression.

Here, we consider yet another application of Markov models: generating artificial text with sufficient spatial dependence to capture the style of the text on which it was trained. In this problem, you will design and implement a Markov model that is capable of generating stylized text, where the style is determined by the input text used to train the model. The result 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 fixed probability no matter where it occurs in the text, but that the probability can depend on the previous k characters. These probability values are parameters of the model, and we can perform parameter estimation to learn the probability values from observed data (this is what we mean by 'training' the model).

In short, we can train a Markov model to generate text in a certain style according to the style of the input text used to estimate the parameters of the model. 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 counts plus pseudocounts).[1] In particular, you can estimate the parameters of a Markov model of order k by 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 are of type `the`, 25 of type `thi`, 20 of type `tha`, and 5 of type `tho`, a second order Markov model

[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 a thought exercise for now; can you think how?)

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. 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.

Here are the steps for training a Markov model on an input text and then generating artificial text from the trained model:

Develop

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

We have begun to implement this step in the Python file `language.py`. The code we provide reads the order k of the Markov model from the command line, reads a text string from a file, and calls a function called `build_dict`, which you will need to write.

a) Write the function `build_dict` that 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 part of the problem, your function should not count the very last k-tuple in the text (here `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 shouldn't have any assumptions hard-coded in about the string that is provided as input. So, for example, you shouldn't assume that you would only ever see lowercase letters and spaces.

This also means that 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 necessary to estimate transition probabilities

Given a k-character key, you must be able to efficiently access the frequency counts of each of the characters that follow it in the training text. This operation is at the crux of the matter, since you will need it to estimate the transition probabilities used to generate random characters in accordance with the Markov model.

b) Write the code for the `collect_counts` function that extends the dictionary with more information. Specifically, instead of just an integer number of occurrences as the value for each key, you'll want to also keep track of how many of those occurrences are associated with each character that follows an occurrence in the training text. 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	followers
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

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

Use the dictionary from `collect_counts` in the previous subproblem to generate artificial text according to the order k Markov model.

c) 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.

We have written code that can use this function to generate characters, starting with the first k characters from the original text, and then repeatedly generating successive pseudo-random characters. Using the example string above, `generate_next_character("gg")` would return `"c"` or `"g"`, each with probability 1/2. After you generate a character, then we shift over one character position, always using the last k characters generated to determine the probabilities for the next character. For example, if your program chooses `c` in the example above, then it would next call `generate_next_character("gc")` and return a `"g"` with probability 2/3 and an `"a"` with probability 1/3.

The program will continue the process until it has generated a total of M characters, and then it will start over and generate another artificial text, and will keep doing it `num_outtext` times.

Check

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 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. `hmm.heart.of.darkness.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. `hmm.paradise.lost.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 punctuation, generated an input text which retained every occurrence of a period. Thus, after cleaning, the input 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 potential extra challenge: This same basic strategy could 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 how you train it.

Problem 2: The occasionally dishonest casino (40 points)

List of files to submit

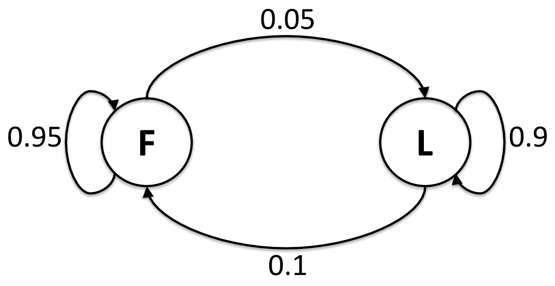
1. `README.problem2.[txt/pdf]`
2. `generate_HMM.py`
3. `dice_roll_sequence.txt`

Plan

In Problem 1, you trained a Markov model on some texts written by Joseph Conrad and John Milton, and then you used the model to generate new text. Now imagine a hidden Markov model (HMM) with two states: 1 and 2. In this system, state 1 may be a MM trained on Conrad's prose while state 2 may be a MM trained on Milton's poetry. The system might stochastically jump from one state to another while it is generating a long piece of text. Generation of every character depends on the current state of the system, Conrad or Milton. The current state in turn depends on the previous state of the system. Using this model to generate text, you may thus end up with text where some sections read like Conrad while other sections read like Milton; there may be abrupt transitions between the two styles along the way, although sometimes it may not be clear exactly where a transition occurred.

Alternatively, and a bit more simply, imagine an occasionally dishonest casino that uses two kinds of dice in order to ensure that, at least in the long run, the house always wins. When beginning a game, the casino picks one die at random and rolls it. Then, before the next roll, the casino either keeps the same die or switches to the other one. This process continues indefinitely with the stipulation that, before every roll, the decision to keep the same die or to switch dice depends on the die used by the casino for the previous roll (since, in order to avoid suspicion, the casino doesn't want to be constantly switching between dice).

Specifically, the probability of rolling each of the numbers 1 to 6 for the fair die used by the casino is 1/6. However, their loaded die has probability 1/2 for the number 6, and 1/10 for all the numbers 1 to 5. Meanwhile, the probabilities of switching between the fair die (state F) and the loaded die (state L) are given below:



Note: In this problem, the term *state* refers to the die being used at any given time and the term *transition probability* is used to describe the probability of switching (or not switching) to the other die (i.e., the probability of making a *transition* from one state to another). Additionally, a series of states is called a *path* and is denoted by Π. Meanwhile, the number rolled using a given die is referred to as an *observation* and the probability of rolling such a number is called an *emission probability.* A series of observations is denoted using X.

a) Enumerate all the possible paths of length 4 and compute the probability of each, assuming the initial state is equally likely to be F or L.

b) Assuming the initial state is equally likely to be F or L, compute the probability of going through the sequence of states $FFLL$ and observing the rolls 1662. Is this probability the same as the probability of the path being $FFLL$, given that you observed the rolls 1662? Why or why not?

c) Which of the two probabilities $P(\Pi = FFLL|X = 1662)$ and $P(\Pi = FLLF|X = 1662)$ is larger? How do you interpret the relation 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 relation?)

d) What path Π (out of all possible paths) do you think is the most likely given the observations 1662? (You don't have to give an exact answer. Just use your intuition to explain how you arrive at an answer.)

e) Let's assume the casino is using the fair die at some point in time. On the next roll (perhaps switching dice, perhaps not), they roll a 6. Is it more likely that the casino switched to the loaded die, or that they continued to use the fair die? And what if they roll two consecutive 6's: what are the probabilities that the last two states were FF, FL, LF, or LL?

Develop

f) Given the state transition diagram for an HMM as drawn above (we have provided these HMM parameters in a file called `HMM.dice.txt`), complete the Python script `generate_HMM.py` to generate a sequence of length 100,000. Your code will be somewhat similar to the code you wrote in Problem 1, but now it needs to account for a hidden state at each position that evolves with Markovian transition probabilities and emits output according to state-dependent emission probabilities.

For each position in the sequence, your model should either transition between different states or remain in the same state according to the given transition probabilities. When the particular state is determined, the model should then emit a die roll (1–6) according to the emission probabilities of the state it is in at that position. Print and save the sequence of underlying states at each position, along with the sequence of 100,000 dice rolls, in a text file named `dice_roll_sequence.txt`. Your output should be formatted to look like this:

```
LLLLFFFFFFL...
66365463216...
```

g) Using your observed sequences of hidden states and dice rolls returned by `generate_HMM` as input, write a function called `audit_casino` that will take as input any given sequence of rolls, and will calculate and return the counts of the various hidden state sequences that prevailed when that sequence of rolls was observed in the output.

Note: Your code shouldn't have any assumptions hard-coded in about the type of value generated by your HMM. So, for example, you shouldn't assume that you would only ever see numbers.

Check

h) Use `audit_casino` to **report** the frequency of each hidden state sequence that ever generated the observed roll sequence 1662 within your longer roll sequence of length 100,000.

Reflect

i) For the specific hidden state sequences FFLL and FLLF, compare their frequencies of prevailing when the rolls 1662 were observed. Are these results as expected, based on your results from part (c)? Why or why not?

j) What are the most frequent state sequences that result in an observed roll sequence of 1662? Do these results make sense? Why or why not?

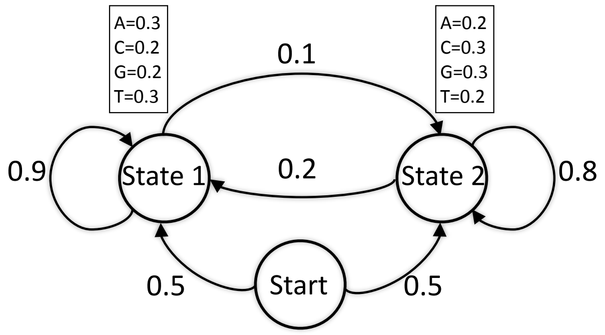
Problem 3: Generating an artificial genomic sequence (20 points)

List of files to submit

1. `README.problem3.[txt/pdf]`
2. `FLAG.txt`
3. `generate_HMM.py`
4. `generate_sequence.py`
5. `artificial_genome.txt`

Plan

To transition back from English literature and dishonest casinos and bring things a little bit closer to the topic of computational genomics, let us imagine that states 1 and 2 correspond to regions of the genome that are more AT rich or more CG rich, respectively. The hidden Markov model will be used to generate an artificial genomic sequence, stochastically alternating between the two kinds of regions.



a) Provide a guess about how long the system should remain in each of the two states on average, and explain the intuition behind your guess.

Develop

b) Use your `generate_HMM` function from problem 2 and the state transition diagram as drawn above (these parameters are specified in a file called `HMM.sequence.txt`) to generate an artificial genomic sequence of length 1000. Print and save your genomic sequence of 1000 nucleotides, as well as the sequence of underlying states at each position in a text file named `artificial_genome.txt`. Note that you will need to convert the state names to single characters to line up with the observed nucleotide sequence. Your output should be formatted to look like this:

```
1211211212...
AGTCGGCAGT...
```

[*Extra challenge:* You could rework your `generate_HMM.py` code so that it generates mixed Conrad/Milton text as we described at the beginning of the question. I'd be curious how that turns out!]

Check

c) Compute how long the system remains in each of the two states on average for the sequence you generated.

Note: It might be helpful for your computations to output your state sequence in the following format:

State	Number of positions spent in that state
1	4
2	6
1	3
2	9
1	2
⋮	⋮

Reflect

d) *Compare your answers from parts (a) and (c) and comment.*

e) Analyze the average distribution of nucleotides that occur in each of the two states. *How closely do your observed distributions match the expected distributions defined by the HMM? Are there regions in your artificially derived sequence that were generated in one state but have observed nucleotide frequencies that look like the other state? Comment.*