UCSC BME 205 Fall 2013

Bioinformatics: Models and Algorithms
Python Assignment 2

(Last Update: 13:22 PST 17 November 2013)

Simple Markov chains
Due Friday 25 Oct 2013 (beginning of class).

This assignment is intended to deepen your understanding of first-order Markov chains and of relative entropy. I will provide you with a data set consisting of many protein sequences in a FASTA file (see FAST/FASTQ assignment). You will build stochastic models of these sequences as zero-order and first-order Markov chains, and measure the information gain of the first-order model over the zero-order model.

I have provided two FASTA files: Python2_f10_1.seqs and Python2_f10_2.seqs for training and testing the models. You will do 2-fold cross-validation (training on each of the sets and testing on the other). These can be accessed on School of Engineering machines directly from the directory /soe/karplus/.html/bme205/f13/markov_files/ so that you don't need to make your own copies.

The combined set was derived from Dunbrack's culledpdb list of X-ray structures with resolution 1.6 Angstroms or better and R-factor <= 0.25, reduced so that no two sequences had more than 90% identity. I used only those sequences that were in our t06 template library, which primarily means that short sequences were removed. I randomly split the list of chain ids into two partitions. For full details on how the set was created, see the Makefile.

We will be modeling the sequences including a special "stop" character at the end of the sequences. This means that the Markov chains will have probabilities that add to one over all strings that end with the stop character, rather than over strings of a fixed length. For simplicity, we'll be using different non-sequence characters to indicate the start of a sequence (^) and end of the sequence($). Note that the training and test data files do not have these special characters—the alphabet is amino acids (with a possibility of special wild cards 'X', 'B', and 'Z').

It might be a good idea in your code to have separate parameters for start and stop characters, and to allow `start=None` to mean that k-mers that start before the first character of the sequence are not counted (similarly for `stop=None`.

The simplest model we discussed in class is the i.i.d. (independent, identically distributed) model, or zero-order Markov model. It assumes that each character of the string comes from the same *background* distribution P_0, and the probability of a string is just the product of the probabilities of the letters of the string (including the final "$"). To estimate the background distribution, we need counts of all the letters in the training set.

The next simplest model was a Markov chain, in which the probability of each character depends only on the preceding character. To estimate the conditional probabilities, we'll need the counts for all pairs of letters in the training set, including initial pairs (like "^M") and ending pairs (like "K$").

In general, for estimating a k-th order Markov chain, we need counts of all (k+1)-mers in the training data.

First program: count-kmers

Write a program called "count-kmers" that reads a FASTA file from stdin and outputs a table of counts to stdout. The output should have two tokens on each line: the k-mer followed by the count of that k-mer. Initial and final k-mers should be symmetrically included, that is, if the sequence is ABDEF and you are counting 3-mers, the counts should be

```
^^A 1
^AB 1
ABD 1
BDE 1
DEF 1
EF$ 1
F$$ 1
```

Note: case should be ignored in the input sequence, so that "abDE", "ABDE", and "abde" are all counted as the same 4-mer.

Hint: there are two different approaches to counting k-mers. In one, you process the file a line at a time (to distinguish id lines), then process the sequence lines a character at a time, using something like `kmer = kmer[0:-1]+letter`. In the other approach, you can define a generator function in Python to read a fasta file one sequence at a time, then extract the kmers from the sequence with something like

```
    for (fasta_id,comment,seq) in read_sequence(genome):
        for start in range(len(seq)-k):
            counts[seq[start:start+k]] += 1
```

Both approaches are useful. The first one does not require large amounts of RAM when you get very long sequences (like whole plant chromosomes), and the second one can be used for other FASTA reading, where the sequences need to be processed individually in more complex ways. Given that you have already written a FASTA parser, I expect that most students will choose to use it.

The "count-kmers" program must accept two options:

`--order` (or -o)
: that gives the order of the model (hence `--order==0` should count 1-mers) and

`--alphabet` (or -a)
: which specifies which letters of the sequence are to be used (for example, to count only standard amino acids, use `--alphabet ACDEFGHIKLMNPQRSTVWY`). All characters in sequences that are not in the alphabet should be ignored. Note: you may want to use the "set" type to represent the alphabet internally. The alphabet should default to ASCII letters if not specified.

The command-line option parsing should be done with the "argparse" module. The actual counting of k-mers should be done by a function you define in a separate module (Markov.py or markov.py). The counts are most easily kept and returned in a "collections.Counter" object, which provides a very convenient way to do all sorts of counting.

If the k-mer counting function is passed sys.stdin as an argument, then you can easily reuse it on other files. Furthermore, if you use `for line in input_stream:` to read the input, then you can do testing by passing in a list of lines:

```
>>> dummy_in= '>seq1\\nABCDEF\\n>seq2\\nAC\\nde\\nFG\\n>seq3\\n'
>>> count=get_counts(dummy_in.splitlines(True), 2, alphabet="ABCDEFGHIJK", start='^', stop='$')
```

Here is an example of what you should get when your run your program:

```
count-kmers -o0 --alphabet=ACDEFGHIKLMNPQRSTVWY < /soe/karplus/.html/bme205/f13/markov_files/Python2_f10_1.seqs
```

$ 2517 A 48441 C 7676 D 34223 E 36355 F 22567 G 45576 H 15831 I 31216 K 31943 L 49311 M 13032 N 25218 P 27188 Q 21205 R 27749 S 35801 T 31893 V 39892 W 8481 Y 20362

Similarly, when you run

```
count-kmers -o1 --alphabet=ACDEFGHIKLMNPQRSTVWY < /soe/karplus/.html/bme205/f13/markov_files/tiny.seqs
```

you should get

```
A$ 1
AC 4
AG 1
AM 7
CA 3
CD 4
DE 4
EF 4
FA 3
FG 1
GG 9
GH 1
GM 1
HI 1
IK 1
K$ 1
M$ 1
MA 6
MC 3
MM 2
MW 1
WM 1
WW 3
^A 1
^M 2
```

For symmetry, report as many stop characters as you report start characters (this makes it easier to make "reverse Markov" models that model the sequences in reverse order).

```
count-kmers -o2 --alphabet=ACDEFGHIKLMNPQRSTVWY < /soe/karplus/.html/bme205/f13/markov_files/tiny.seqs
```

you should get

```
A$$ 1
ACD 4
AGH 1
AM$ 1
AMA 4
AMC 1
```

```
AMM 1
CAM 3
CDE 4
DEF 4
EFA 3
EFG 1
FAC 2
FAG 1
FGG 1
GGG 8
GGM 1
GHI 1
GMA 1
HIK 1
IK$ 1
K$$ 1
M$$ 1
MA$ 1
MAC 1
MAM 4
MCA 3
MMM 1
MMW 1
MWW 1
WMC 1
WWM 1
WWW 2
^AC 1
^MA 1
^MC 1
^^A 1
^^M 2
```

Note: the output should be sorted in alphabetical order, with a single space separator, so that I can check I/O behavior using the "diff" program.

Optional bonus points:

- Use the doctest module to do unit testing for the functions in your markov module.
- Add options `--start` and `--stop` to change the start and stop characters.
- Add an option to sort the k-mers in reverse order of counts for the output file.
- Check that the alphabet has only printable, non-whitespace characters in it. Raise an exception if there are problems.
- Check that the stop and start characters are not part of the alphabet, and raise an exception if they are.

Second program: coding-cost

Write a program that reads in a table of k-mer counts (in the same format as you output them) from stdin and FASTA-formatted sequences from a file (passed as a required argument to the program), and computes the total encoding cost in bits for the sequences. The encoding cost of a character x using model M is -log_2 P_M(x), and the encoding cost of a sequence is the sum of the costs for the characters (note: this is equivalent to taking -log_2 of the product of the probabilities).

Report not only the total encoding cost for the file, but the average cost per sequence and per character. So that everyone gets the same numbers, for the per-character costs the numerator is the total cost, which includes encoding exactly one stop character at the end (for any order), and the denominator is the number of characters *not* counting the stop character. I'm not defending this as necessarily the best numerator and denominator to use for average cost/character, but it is the choice I used in generating the "right" answers.

Note that to do the computation, you need to convert the counts into probabilities. The simplest approach is to normalize the counts to sum to 1 over each "condition". Note that `kmer[0:-1]` is the "condition" for a given k-mer. You might think that because the FASTA training file will be large, you will not need to worry about pseudocounts, but even by order 2, there are 3-mers that have zero counts, so the maximum-likelihood estimate of their probability is 0 and the encoding cost is infinite (resulting in a "ValueError: math domain error" from python if you actually try to compute it).

The next simplest approach is to add a pseudocount to every count (including the zero counts for a context, so if you have seen "WWA", "WWC", ... but not "WWH" or "WWI", you have the context "WW" and will need to add a pseudocount for every "WWx" where x ranges over your alphabet. I recommend adding a function to your markov module that produces a conditional probability dict from a count dict or Counter (without changing count), so that you can easily modify the way you handle the probability estimation. The default pseudocount should be 1.0 for each cell.

The simple approach of making sure that pseudocounts have been used for letters in every observed context will work fine up to about order 2, but for higher order Markov chains, you might run into k-mers for which even the k-1-mer that is the context has never been seen in the training set. (For example, Python2_f10_2.seqs has a sequence starting with CYC, producing 4-mer ^CYC , but the context ^CY never occurs in Python2_f10_1.seqs.)

You can fix this problem by making sure that the log-prob table has keys for all k-mers over the alphabet.

One tricky point in the log-prob table: We know that once we get to the end of a sequence and have seen the final "$", then all subsequent letters must be "$", so we do not want P("A$A") to be non-zero, even if we have seen counts for "A$$".

Hint: for this assignment, since we are not looking at the encoding cost individually for each sequence, but only the total encoding cost, you can summarize the fasta file with the same sort of dictionary of counts that was used for count-kmers. You can determine the size of k-mer you need to count by looking at the size of the keys you read in for the model.

Run coding_cost on both sets of sequences using both order 0 and order 1 models. Get the coding costs both for the training set and for the test set. For comparison, my code with an autodetected alphabet got

Train	Test	order 0 bits/char	order1 bits/char
Python2_f10_1	Python2_f10_1	4.22893	4.20759
Python2_f10_2	Python2_f10_1	4.22897	4.20872
Python2_f10_2	Python2_f10_2	4.22678	4.20476
Python2_f10_1	Python2_f10_2	4.22686	4.20601

The table with the constrained alphabet ACDEFGHIKLMNPQRSTVWY is

Train	Test	order 0 bits/char	order1 bits/char
Python2_f10_1	Python2_f10_1	4.22796	4.20691
Python2_f10_2	Python2_f10_1	4.22803	4.20802
Python2_f10_2	Python2_f10_2	4.22590	4.20424
Python2_f10_1	Python2_f10_2	4.22597	4.20536

Note that a simple uniform model over 20 characters would take 4.32193 bits/char, and over a 21-letter alphabet (including stop) would take 4.39232 bits/char.

Optional bonus points:

- Autodetect the alphabet. That is, set the alphabet to any letters, or maybe even any non-whitespace, characters that occur in the sequences.
- For each of the conditional probability distributions for the possible prior states (for first-order, this would be 21 states: start and 20 amino acids), compute the relative entropy between it and the background distribution. Where are the biggest information gains coming from?
- Test the "reversibility" of an alphabet, by determining the coding cost for reversed sequences using the Markov chain. Note that since we counted off-the-end characters symmetrically at the beginning and end, we can simply create a new reverse_count object which has the same values but with each key reversed (and stop/start codes appropriately swapped).
- Have the program automatically run tests of different values of the pseudocount, to find out what value works best. This can be as simple as

  ```
  for pseudocount in [1e-10, 0.01, 0.1, 0.2, 0.5, 1, 2, 5, 10, 100]:
  ```

  or as sophisticated as you want to make it.
- Try a more sophisticated regularizer than pseudocounts. For example, you could use the probabilities from a smaller context. One method I think is cool, but have not yet implemented, is to have the pseudocounts for a order k+1 model be alpha*probabilities from an order k model, recursively down to pseudocounts of alpha/num_letters for an order 0 model.
- Look at the encoding costs for sequences over other alphabets: I have provided a couple of other sets, one using the str2 secondary-structure alphabet, and one using the near-backbone-11 burial alphabet: Python2_f10_1.str2s Python2_f10_2.str2s Python2_f10_1.near-backbone-11s Python2_f10_2.near-backbone-11s The str2 alphabet includes the following "normal" characters: ABCEGHMPQSTYZ. The character X is also possible (as a wildcard) and I (as an alias for H). The near-backbone-11 burial alphabet has letters ABCDEFGHIJK.
- To get even fancier with alphabets, allow the user to specify the alphabet by name, and look it up in the /projects/compbio/lib/alphabet/ files we have defined for our various other programs. The ExtAA alphabet gives the definition of the amino acids. Note the specification of wildcards. Another alphabet we use a lot is the EHL2 alphabet, in DSSP.alphabet---note its use of aliases to collapse several different letters in a sequence into the same equivalence class.
- Think about adding options to handle letters that are not the 20 standard amino acids (wildcards B,Z, and X, and spaces "." and "-"). The basic assignment calls for you to ignore any letters or punctuation not in the alphabet you specify.
  - X is a code for any amino acid
  - B is a code for aspartic acid or asparagine (D or N)

- Z is a code for glutamic acid or glutamine (E or Q)
- I have heard that mass spectrometry people sometimes use J for isoleucine or leucine (I or L), since they have identical mass, but I've not seen this in files I've used.

The B and Z codes occur mainly in protein sequences that have been determined by old-fashioned chemical methods, not from translating DNA sequences.

There are several ways to handle wildcards and non-standard amino acid letters:

1. treat them the same as spaces, tabs, ".", and "-". That is ignore them completely on input. This will make one erroneous transition: ABF would look like AF. This is the default behavior that your programs are expected to provide.
2. Arbitrarily pick one of the possible amino acids (maybe replace B by D and Z by E).
3. Treat the character as a gap in the sequence—handle the subsequences on each side as different sequences. There are two variants of this—one which starts a new sequence after the B like any other sequence (so ABF would have a start->F transition) and one that doesn't add the start->F transition. Note that adding a start->F transition may throw off probabilities for what the first residue of a sequence really is.

There are undoubtedly other methods also. What ever you do **document** your choices.

To turn in

I want copies of your programs on paper, but I don't want all the output from every run of the programs! Instead, write a short summary describing the results of the tests, giving the average encoding costs for each set of sequences for each model. Try to write your summary as if you were describing the method and the results in a journal paper that has a page limit: you want to describe the data sets and methods completely but concisely. If you do it well, the presentation should take less than a page.

Turn in your summary page with the program stapled after it. (Avoid paper clips—they have a tendency to either come off or to pick up other people's papers.)

I also want a directory name for a directory on the School of Engineering machines that I can read that contains all the programs and writeup for this assignment. Make sure that the directory and all files in it are publicly readable, and that all directories on the path to it are publicly executable, or I won't be able to access the files.

Hints

- The programs specified here do not need to really read fasta as sequences. The id lines can be mostly ignored and the sequences processed a character at a time (you do need to know when to start a new sequence). But you can use your FASTA parser if you want to.
- Uppercase your lines from the fasta input, since you want to ignore case.
- Because real sequences may have length 0, be sure your program can handle the empty sequence properly.
- The reversal of a string is standard python function, but it doesn't return a string object, but a "reversed" object, which appears to be derived from lists rather than strings. To get a reversed string for kmer, use

```
    "".join(reversed(kmer))
```

- An observation from previous years: A lot of students managed to make their programs more complex to write and harder to use by insisting on file I/O instead of using stdin and stdout. The assignment **requires** that you use stdin and stdout. Also, error messages should be printed to stderr (to avoid corrupting the main output in stdout).
- Remember that Markov chains are *conditional* probabilities P(x_i=s | x_{i-1}=t), not joint probabilities.
- Break code into paragraphs, separated by blank lines, and start each paragraph with a comment as a topic sentence for the paragraph.
- Don't read all the data before processing any. If you want to make a fasta-reading module, make sure its interface returns one sequence at a time, using the `yield` statement in Python to make a generator.
- A sequence may span many lines. Make sure that you don't start a new sequence with each new line. Sequences only start after an ID line, which starts with ">".
- Don't have a logarithm computation in the inner loop. Convert your counts into costs (negative log probabilities) before running through the test set.
- Make sure that your function definitions all have docstrings.

Things learned for next year

Students were very poor at following I/O specs this year—many did not read from stdin as required, and many got the program names wrong. I also allowed too much variability: some students required `--alphabet`; some did not allow it. some students even required `--order` on coding-cost, where it should be trivially deduced from the input.

I'll probably have to provide more explicit examples of the command lines I'll run, since students seem incapable of reading the specs.

I failed to specify that fasta files should be positional arguments for coding-cost, so many required a `--in` or`--fasta` argument.

SoE home

Kevin Karplus's home page

Biomolecular Engineering Department

BME 205 home page

UCSC Bioinformatics research

Questions about page content should be directed to Kevin Karplus
Biomolecular Engineering
University of California, Santa Cruz
Santa Cruz, CA 95064
USA
karplus@soe.ucsc.edu
1-831-459-4250
318 Physical Sciences Building