B551 Assignment 3: Probability

Fall 2018
Due: Sunday November 25, 11:59PM
(You may submit up to 48 hours late for a 10% penalty.)

This assignment will give you a chance to practice probabilistic inference for some real-world problems.

You'll work in a group of 1-3 people for this assignment; we've already assigned you to a group (see details below) based on the input you provided with your A2 team feedback. We tried to accommodate as many of your requests as possible, but we could not satisfy all of them. You should only submit **one** copy of the assignment for your team, through GitHub, as described below. All people on the team will receive the same grade on the assignment, except in unusual circumstances; we will collect feedback about how well your team functioned in order to detect these circumstances. Please read the instructions below carefully; we cannot accept any submissions that do not follow the instructions given here. Most importantly: please **start early,** and ask questions on Piazza or in office hours. Note that all groups will do the same assignment regardless of how many people are in the team, but obviously we will implicitly have higher expectations for quality and completeness for larger teams.

The following problems require you to write programs in Python. You may import standard Python modules for routines not related to AI, such as basic sorting algorithms and basic data structures like queues. You must write all of the rest of the code yourself. If you have any questions about this policy, please ask us. We recommend using the CS Linux machines (e.g. `burrow.soic.indiana.edu`). You may use another development platform (e.g. Windows), but make sure that your code works on the CS Linux machines before submission. For each programming problem, please include a detailed comments section at the top of your code that describes: (1) a description of how you formulated the problem, including precisely defining the abstractions (e.g. HMM formulation); (2) a brief description of how your program works; (3) a discussion of any problems, assumptions, simplifications, and/or design decisions you made; and (4) answers to any questions asked below in the assignment.

Academic integrity. You and your teammates may discuss the assignment with other people at a high level, e.g. discussing general strategies to solve the problem, talking about Python syntax and features, etc. You may also consult printed and/or online references, including books, tutorials, etc., but you must cite these materials (e.g. in source code comments). However, the work and code that you and your partners submit must be your group's own work, which your group personally designed and wrote. You may not share written answers or code with any other students except your own partner, nor may you possess code written by another student who is not your partner, either in whole or in part, regardless of format.

Part 0: Getting started

You can find your assigned teammate by logging into IU Github, at `http://github.iu.edu/`. In the upper left hand corner of the screen, you should see a pull-down menu. Select `cs-b551-fa2018`. Then in the yellow box to the right, you should see a repository called *userid1-userid2-userid3*-a3, where the other user IDs correspond to your teammate(s).

To get started, clone the github repository:

git clone https://github.iu.edu/cs-b551-fa2018/*your-repo-name*-a3

where *your-repo-name* is the one you found on the GitHub website above.

Part 1: Part-of-speech tagging

Natural language processing (NLP) is an important research area in artificial intelligence, dating back to at least the 1950's. One of the most basic problems in NLP is *part-of-speech tagging*, in which the goal is to mark every word in a sentence with its part of speech (noun, verb, adjective, etc.). This is a first step towards extracting semantics from natural language text. For example, consider the following sentence:

Her position covers a number of daily tasks common to any social director.

Part-of-speech tagging here is not easy because many of these words can take on different parts of speech depending on context. For example, *position* can be a noun (as in the above sentence) or a verb (as in "They position themselves near the exit"). In fact, *covers*, *number*, and *tasks* can all be used as either nouns or verbs, while *social* and *common* can be nouns or adjectives, and *daily* can be an adjective, noun, or adverb. The correct labeling for the above sentence is:

Her	position	covers	a	number	of	daily	tasks	common	to	any	social	director.
DET	NOUN	VERB	DET	NOUN	ADP	ADJ	NOUN	ADJ	ADP	DET	ADJ	NOUN

where DET stands for a determiner, ADP is an adposition, ADJ is an adjective, and ADV is an adverb.[1] Labeling parts of speech thus involves an understanding of the intended meaning of the words in the sentence, as well as the relationships between the words.

Fortunately, some relatively simple statistical models can do amazingly well at solving NLP problems. In particular, consider the Bayesian network shown in Figure 1(a). This Bayes net has a set of N random variables $S = \{S_1, \ldots, S_N\}$ and N random variables $W = \{W_1, \ldots, W_N\}$. The W variables represent observed words in a sentence, so $W_i \in \{w | w \text{ is a word in the English language}\}$. The variables in S represent part of speech tags, so $S_i \in \{\text{VERB}, \text{NOUN}, \ldots\}$. The arrows between W and S nodes model the probabilistic relationship between a given observed word and the possible parts of speech it can take on, $P(W_i|S_i)$. (For example, these distributions can model the fact that the word "dog" is a fairly common noun but a very rare verb. The arrows between S nodes model the probability that a word of one part of speech follows a word of another part of speech, $P(S_{i+1}|S_i)$. (For example, these arrows can model the fact that verbs are very likely to follow nouns, but are unlikely to follow adjectives.)

We can use this model to perform part-of-speech tagging as follows. Given a sentence with N words, we construct a Bayes Net with $2N$ nodes as above. The values of the variables W are just the words in the sentence, and then we can perform Bayesian inference to estimate the variables in S.

[1]If you didn't know the term "adposition", neither did I. The adpositions in English are prepositions; in many languages, there are postpositions too. But you won't need to understand the linguistic theory between these parts of speech to complete the assignment; if you're curious, you might check out the "Part of Speech" Wikipedia article for some background.

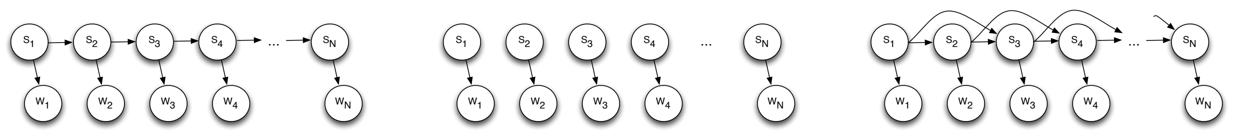

Figure 1: Three Bayes Nets for part of speech tagging: (a) an HMM, (b) a simplified model, and (c) a more complicated model.

Data. To help you with this assignment, we've prepared a large corpus of labeled training and testing data, consisting of nearly 1 million words and 50,000 sentences. The file format of the datasets is quite simple: each line consists of a word, followed by a space, followed by one of 12 part-of-speech tags: ADJ (adjective), ADV (adverb), ADP (adposition), CONJ (conjunction), DET (determiner), NOUN, NUM (number), PRON (pronoun), PRT (particle), VERB, X (foreign word), and . (punctuation mark). Sentence boundaries are indicated by blank lines.[2]

What to do. Your goal in this part is to implement part-of-speech tagging in Python, using Bayes networks.

1. First you'll need to estimate the probabilities of the HMM above, namely $P(S_1)$, $P(S_{i+1}|S_i)$, and $P(W_i|S_i)$. To do this, use the labeled *training* file we've provided.

2. Your goal now is to label new sentences with parts of speech, using the probability distributions learned in step 1. To get started, consider the simplified Bayes net in Figure 1(b). To perform part-of-speech tagging, we'll want to estimate the most-probable tag s_i^* for each word W_i,
$$s_i^* = \arg\max_{s_i} P(S_i = s_i|W).$$
Implement part-of-speech tagging using this simple model. *Hint:* This is easy; if you don't see why, try running Variable Elimination by hand on the Bayes Net in Figure 1(b).

3. Now consider the Bayes net of Figure 1(a), which is a richer model that incorporates dependencies between words. Implement the Viterbi algorithm to find the maximum a posteriori (MAP) labeling for the sentence – i.e. the most likely state sequence:
$$(s_1^*, \ldots, s_N^*) = \arg\max_{s_1,\ldots,s_N} P(S_i = s_i|W).$$

4. Consider the Bayes Net of Figure 1c, which is a better model of language because it incorporates some longer-term dependencies between words. It's no longer an HMM, so one can't use Viterbi, but we can use MCMC. Write code that uses MCMC to sample from the

[2]This dataset is based on the Brown corpus. Modern part-of-speech taggers often use a much larger set of tags – often over 100 tags, depending on the language of interest – that carry finer-grained information like the tense and mood of verbs, whether nouns are singular or plural, etc. In this assignment we've simplified the set of tags to the 12 described here; the simple tag set is due to Petrov, Das and McDonald, and is discussed in detail in their 2012 LREC paper if you're interested.

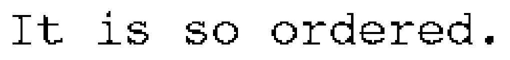
It is so ordered.

Figure 2: In OCR, our goal is to extract text from a noisy scanned image of a document.

posterior distribution of Fig 1c, $P(S|W)$, after a warm-up period, and shows five sampled particles. Then estimate the best labeling for each word (by picking the maximum marginal for each word, $s_i^* = \arg\max_{s_i} P(S_i = s_i|W)$, as in step 2). (To do this, just generate many (thousands?) of samples and, for each individual word, check which part of speech occurred most often.)

Your program should take as input two filenames: a training file and a testing file. The program should use the training corpus for Step 1, and then display the output of Steps 2-4 on each sentence in the testing file. For the result generated by each of the three approaches (Simple, HMM, and Complex), as well as for the ground truth result, your program should output the logarithm of the posterior probability for each solution it finds under each of the three models in Figure 1. It should also display a running evaluation showing the percentage of words and whole sentences that have been labeled correctly according to the ground truth so far. For example:

```
[djcran@raichu djc-sol]$ ./label.py training_file testing_file
Learning model...
Loading test data...
Testing classifiers...
                  Simple     HMM Complex Magnus     ab integro seclorum nascitur ordo .
0. Ground truth   -48.52  -64.33  -78.21   noun   verb     adv     conj     noun noun .
      1. Simple   -47.29  -66.74  -79.01   noun   noun    noun      adv     verb noun .
         2. HMM   -47.48  -63.83  -79.12   noun   verb     adj     conj     noun verb .
     3. Complex   -48.52  -64.33  -78.21   noun   verb     adv     conj     noun noun .

==> So far scored 1 sentences with 17 words.
                   Words correct:     Sentences correct:
   0. Ground truth:      100.00%             100.00%
     1. Simplified:       42.85%               0.00%
        2. HMM MAP:       71.43%               0.00%
   3. Complex MCMC:      100.00%             100.00%
```

We've already implemented some skeleton code to get you started, in three files: label.py, which is the main program, pos_scorer.py, which has the scoring code, and pos_solver.py, which will contain the actual part-of-speech estimation code. You should only modify the latter of these files; the current version of pos_solver.py we've supplied is very simple, as you'll see. In your report, please make sure to include your results (accuracies) for each technique on the test file we've supplied, `bc.test`.

Part 2: Optical Character Recognition (OCR)

To show the versatility of HMMs, let's try applying them to another problem; if you're careful and you plan ahead, you can probably re-use much of your code from Part 1 to solve this problem. Our goal is to recognize text in an image – e.g., to recognize that Figure 2 says "It is so ordered." We'll consider a simplified OCR problem in which the font and font size is known ahead of time, but the basic technique we'll use is very similar to that used by commercial OCR systems.

Modern OCR is very good at recognizing documents, but rather poor when recognizing isolated characters. It turns out that the main reason for OCR's success is that there's a strong *language model:* the algorithm can resolve ambiguities in recognition by using statistical constraints of English (or whichever language is being processed). These constraints can be incorporated very naturally using an HMM.

Let's say we've already divided a text string image up into little subimages corresponding to individual letters; a real OCR system has to do this *letter segmentation* automatically, but here we'll assume a fixed-width font so that we know exactly where each letter begins and ends ahead of time. In particular, we'll assume each letter fits in a box that's 16 pixels wide and 25 pixels tall. We'll also assume that our documents only have the 26 uppercase latin characters, the 26 lowercase characters, the 10 digits, spaces, and 7 punctuation symbols, `(),.-!?'"`. Suppose we're trying to recognize a text string with n characters, so we have n observed variables (the subimage corresponding to each letter) $O_1, ..., O_n$ and n hidden variables, $l_1 ..., l_n$, which are the letters we want to recognize. We're thus interested in $P(l_1, ..., l_n|O_1, ..., O_n)$. As in part 1, we can rewrite this using Bayes' Law, estimate $P(O_i|l_i)$ and $P(l_i|l_{i-1})$ from training data, then use probabilistic inference to estimate the posterior, in order to recognize letters.

What to do. Write a program called `ocr.py` that is called like this:

```
./ocr.py train-image-file.png train-text.txt test-image-file.png
```

The program should load in the train-image-file, which contains images of letters to use for training (we've supplied one for you). It should also load in the train-text file which is simply some text document that is representative of the language (English, in this case) that will be recognized. (The training file from Part 1 could be a good choice). Then, it should use the classifier it has learned to detect the text in test-image-file.png and output the recognized text on the last line of its output.

```
[djcran@raichu djc-sol]$ ./ocr.py train-image-file.png train-text.txt test-image-file.png
Simple:  1t 1s so orcerec.
Viterbi: It is so ordered.
Final answer:
It is so ordered.
```

Make sure to include a detailed report at the top of your code that explains how your approach works, any design decisions you made, other designs you tried, etc.

Hints. We've supplied you with skeleton code that takes care of all the image I/O for you, so you don't have to worry about any image processing routines. The skeleton code converts the images into simple Python list-of-lists data structures that represents the characters as a 2-d grid of black and white dots. You'll need to define an HMM and estimate its parameters from training data. The transition and initial state probabilities should be easy to estimate from the text training data. For the emission probability, we suggest using a simple naive Bayes classifier. The train-image-file.png file contains a perfect (noise-free) version of each letter. The text strings your program will encounter will have nearly these same letters, but they may be corrupted with noise. If we assume that m% of the pixels are noisy, then a naive Bayes classifier could assume that each pixel of a given noisy image of a letter will match the corresponding pixel in the reference letter with probability $(100 - m)$%. This problem is left purposely open-ended; feel free to try other emission probabilities, Bayes Net structures, and inference algorithms to get the best results.

What to turn in

Turn in the file required above by simply putting the finished version (of the code with comments) on GitHub (remember to `add`, `commit`, `push`) — we'll grade whatever version you've put there as of 11:59PM on the due date. To make sure that the latest version of your work has been accepted by GitHub, you can log into the github.iu.edu website and browse the code online.