B551 Assignment 3: Probability, NLP, and computer vision

Fall 2022
Due: Monday December 5, 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 in natural language processing and computer vision.

Guidelines for this assignment

Coding requirements. For fairness and efficiency, we use a semi-automatic program to grade your submissions. As usual, we require that: 1. You must code this assignment in Python 3; 2. You should test your code on silo.luddy.indiana.edu; 3. Your code must obey the input and output specifications given below. 4. You may import standard Python modules for routines not related to AI, such as basic sorting algorithms and data structures, as long as they are already installed on silo.luddy.indiana.edu; and 5. Make sure to use the program file name we specify.

Groups. You'll work in a group of 1-3 people for this assignment according to your preferences. You should only submit one copy of the assignment for your team, through GitHub. All the people on the team will receive the same grade, except in unusual circumstances; we will collect feedback about how well your team functioned in order to detect these circumstances.

Coding style and documentation. We will not explicitly grade coding style, but it's important that you write your code in a way that we can easily understand it. Please use descriptive variable and function names, and use comments when needed to help us understand code that is not obvious. For each of these problems, you will face some design decisions along the way. Your primary goal is to write clear code that finds the correct solution in a reasonable amount of time. To encourage innovation, we will conduct a competition among programs to see which can solve the hardest problems in the shortest amount of time.

Report. Please put a report describing your assignment in the Readme.md file in your Github repository. For each problem, please include: (1) a description of how you formulated each problem; (2) a brief description of how your program works; (3) and discussion of any problems you faced, any assumptions, simplifications, and/or design decisions you made. These comments are especially important if your code does not work perfectly, since it is a chance to document the energy and thought you put into your solution.

Academic integrity. We take academic integrity very seriously. To maintain fairness to all students in the class and integrity of our grading system, we will prosecute any academic integrity violations that we discover. *Before beginning this assignment, make sure you are familiar with the Academic Integrity policy of the course, as stated in the Syllabus, and ask us about any doubts or questions you may have.* To briefly summarize, you 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). We expect that you'll write your own code and not copy anything from anyone else, including online resources. *However, if you do copy something (e.g., a small bit of code that you think is particularly clever), you have to make it explicitly clear which parts were copied and which parts were your own. You can do this by putting a very detailed comment in your code, marking the line above which the copying began, and the line below which the copying ended, and a reference to the source.* Any code that is not marked in this way must be your own, which you personally designed and wrote. You may not share written answers or code with any other students, nor may you possess code written by another student, either in whole or in part, regardless of format.

Part 0: Getting started

Please fill out the form for team selection on https://forms.gle/ViAnCC6e6tH1dbSR6 by Wednesday, November 16 at 23:59 (please fill out the form even if you want us to assign you to a team randomly). We will publish final team names by Friday, November 18 in the following format: userid1-a2, userid1-userid2-a2, or userid1-userid2-userid3-a2, where the other user ID(s) correspond to your teammate(s). You should submit your code in a repository with the same name as your team name. If you don't already know your team mates, you can write them on userid@iu.edu. To get started, clone the github repository using one of the two commands:

```
git clone git@github.iu.edu:cs-b551-fa2022/a3-release-res
git clone https://github.iu.edu/cs-b551-fa2022/a3-release-res
```

Part 1: Part-of-speech tagging

A basic problems in Natural Language Processing 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.). Sometimes this is easy: a sentence like "Blueberries are blue" clearly consists of a noun, verb, and adjective, since each of these words has only one possible part of speech (e.g., "blueberries" is a noun but can't be a verb).

But in general, one has to look at all he words in a sentence to figure out the part of speech of any individual word. For example, consider the — grammatically correct! — sentence: "Buffalo buffalo Buffalo buffalo buffalo buffalo Buffalo buffalo." To figure out what it means, we can parse its parts of speech:

Buffalo	buffalo	Buffalo	buffalo	buffalo	buffalo	Buffalo	buffalo.
Adjective	Noun	Adjective	Noun	Verb	Verb	Adjective	Noun

(In other words: the buffalo living in Buffalo, NY that are buffaloed (intimidated) by buffalo living in Buffalo, NY buffalo (intimidate) buffalo living in Buffalo, NY.)

That's an extreme example, obviously. Here's a more mundane sentence:

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] Many of these words can be different parts of speech: "position" and "covers" can both be nouns or verbs, for example, and the only way to resolve the ambiguity is to look at the surrounding words. 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, statistical models work amazingly well for NLP problems. Consider the Bayes net shown in Figure 1(a). This Bayes net has random variables $S = \{S_1, \ldots, S_N\}$ and $W = \{W_1, \ldots, W_N\}$. The W's represent observed words in a sentence. The S's represent part of speech tags, so $S_i \in \{\text{VERB}, \text{NOUN}, \ldots\}$. The arrows between W and S nodes model the 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.)

Data. To help you with this assignment, we've prepared a large corpus of labeled training and testing data. Each line consists of a sentence, and each word is 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).[2]

---

[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, check out the "Part of Speech" Wikipedia article for some background.

[2]This dataset is based on the Brown corpus. Modern part-of-speech taggers often use a much larger set of tags – often over
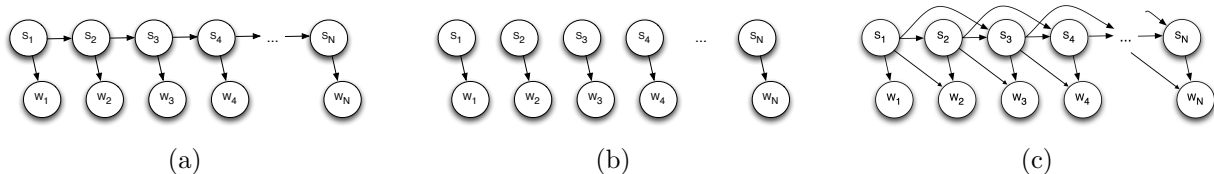
Figure 1: Bayes Nets for part of speech tagging: (a) HMM, and (b) simplified model, and (c) complicated model.

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

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.

2. Now consider Figure 1(a), a richer Bayes net that incorporates dependencies between words. Implement Viterbi to find the maximum a posteriori (MAP) labeling for the sentence,

$$(s_1^*, \ldots, s_N^*) = \arg\max_{s_1,\ldots,s_N} P(S_i = s_i | W).$$

3. Consider the Bayes Net of Figure 1c, which could be a better model because it incorporates richer dependencies between words. But it's not an HMM, so we can't use Viterbi. Implement Gibbs Sampling to sample from the posterior distribution of Fig 1c, $P(S|W)$. Then estimate the best labeling for each word (by picking the maximum marginal for each word, $s_i^* = \arg\max_{s_i} P(S_i = s_i | W)$. (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 a training filename and a testing filename. The program should use the training corpus to estimate parameters, and then display the output of Steps 1-3 on each sentence in the testing file. For the result generated by each of the three approaches (Simple, HMM, Complex), as well as for the ground truth result, your program should output the logarithm of the joint probability $P(S, W)$ 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 so far. For example:

```
[ztiganj@raichu zt-sol]$ python3 ./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   -73.43    noun   verb      adv     conj     noun  noun .
      1. Simplified  -47.29  -66.74   -75.29    noun   noun     noun      adv     verb  noun .
            2. HMM   -47.48  -63.83   -74.12    noun   verb      adj     conj     noun  verb .
   3.    Complex    -47.50  -64.21   -72.02    noun   verb      adv     conj     noun  noun .

==> So far scored 1 sentences with 17 words.
```

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.

```
                Words correct:      Sentences correct:
  0. Ground truth      100.00%              100.00%
    1. Simplified       42.85%                0.00%
          2. HMM        71.43%                0.00%
      3. Complex       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`. Your code should finish within about 10 minutes.

Part 2: Reading text

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." But the images are noisy, so any particular letter may be difficult to recognize. However, if we make the assumption that these images have English words and sentences, we can use statistical properties of the language to resolve ambiguities.

It is so ordered.

Figure 3: Our goal is to extract text from a noisy scanned image of a document.

We'll assume that all the text in our images has the same fixed-width font of the same size. In particular, 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 `image2text.py` that is called like this:

```
python3 ./image2text.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 text training 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, using (1) the simple Bayes net of Figure 1b and (2) the HMM of Fig 1a with MAP inference (Viterbi). The last two lines of output from your program should be these two results, as follows:

```
[ztiganj@tank]$ python3 ./image2text.py train-image-file.png train-text.txt test-image-file.png

Simple: 1t 1s so orcerec.
   HMM: It is so ordered.
```

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 `courier-train.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 $(100 - m)\%$.

What to turn in

Create a private repository using the name you found in the Group Assignment Sheet, and keep the same directory structure as in `a3-release` (i.e., separating the programs into Part1 and Part2). Make sure that you stick to this naming scheme and directory structure so that our autograder can locate your submission correctly. And make sure it is private (not internal or public) so others will not see your submission. Turn in the required programs on GitHub (remember to `add, commit, push`) — we'll grade whatever version you've put there as of 11:59:59PM 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.