# Assignment 4

B551

**Due: Monday November 20, 11:59:59 PM Eastern time**

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 an automatic program to grade your submissions. This means you must write your code carefully so that our program can run your code and understand its output properly. In particular:

1. You must code this assignment in Python 3, not Python 2.
2. Make sure to use the program file name we specify.
3. Use the skeleton code we provide, and follow the instructions in the skeleton code (e.g., to not change the parameters of some functions). 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 like queues, as long as they are already installed on `silo.sice.indiana.edu`.

***Groups***. You'll work in a team of 2 people for **Part 1** in this assignment; we've already assigned you to a team. You can find your team on canvas by looking into people –> Groups. The group having the button ***Visit*** is your assigned group. 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, so we will collect feedback about how well your team functioned.

***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.

***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. Additionally, in your report please describe (1) how you divided the work among team members, (2) contribution of each team member.

# 1 Part 0: Getting started

To get started, clone the Github repository created for this assignment. Type below command in your command line:

```
git clone https://github.iu.edu/cs-b551-fa2023/assignment4.git
```
Or
```
git clone git@github.iu.edu:cs-b551-fa2023/assignment4.git
```

(If neither command works, you probably need to set up IU GitHub ssh keys.)

# 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."

| 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 2. 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 $S_i \in \{$ VERB, NOUN, ...$\}$. 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, $P(W_i \mid S_i)$. (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, $P(S_{i+1} \mid S_i)$. (For example, these arrows can model the fact that verbs are very likely to follow nouns, but are unlikely to follow adjectives.)

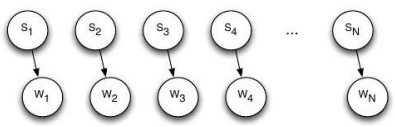


Figure 1: Simplified Model

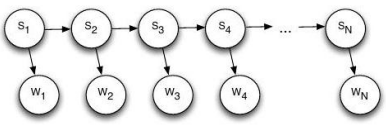


Figure 2: HMM

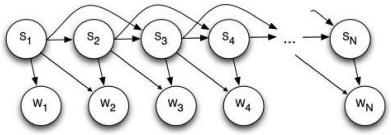


Figure 3: Complicated Model

***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]
***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. To perform part-of-speech tagging, we'll want to estimate the most-probable tag $s_i^*$ for each word $W_i$,

$$s_i^* = \arg\max_{s_i} P(S_i = s_i \mid W)$$

   Implement part-of-speech tagging using this simple model.

[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

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

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

3. Consider the Bayes Net of Figure 3, 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, $P(S \mid W)$. Then estimate the best labeling for each word (by picking the maximum marginal for each word, $s_i^* = \arg\max_{s_i} P(S_i = s_i \mid W)$. (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, 2 and 3. It should also display a running evaluation showing the percentage of words and whole sentences that have been labeled correctly so far. For example:

```
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.

                  Words correct:  Sentences correct:
 0.  Ground truth        100.00%            100.00 %
   1.  Simplified          2.85 %             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-ofspeech 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 4 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. 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

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.

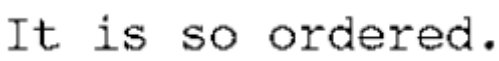


Figure 4: Our goal is to extract text from a noisy scanned image of a document.

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) $O_1, \ldots, O_n$ and $n$ hidden variables, $l_1 \ldots, l_n$, which are the letters we want to recognize. We're thus interested in $P(l_1, \ldots, l_n \mid O_1, \ldots, O_n)$. As in part 1 , we can rewrite this using Bayes' Law, estimate $P(O_i \mid l_i)$ and $P(l_i \mid l_{i-1})$ 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 1 and (2) the HMM of Fig 2 with MAP inference (Viterbi). The last two lines of output from your program should be these two results, as follows:

```
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

For this assignment you will be creating two private repositories.

1. For Part 1, create a private repository on GitHub under cs-b551-fa2023 using the name `userid1-userid2-a4` where the user IDs correspond to the _iu_id of team members sorted alphabetically (e.g., sanagra-sblancor-a4). Make sure that both the team members have access to the repository. You can add your team member by going to your repository –> settings –> Collaborators and teams. Submit your solution for Part 1 using this repository.

2. For Part 2, create a private repository using the name your_iu_id-a4 on GitHub under cs-b551-fa2023, where your_iu_id is the text before the @ sign in your IU email address (e.g., sblancor-a4). Submit your solution for Part 2 using this repository.

Make sure that both of your repositories are 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. Also remember to put your report in the `README.md` file. Note that for this assignment you will be submitting two reports; one for each repository. To make sure that the latest version of your work has been accepted by GitHub, you can log into the `http://github.iu.edu` website and browse the code online. Your programs must obey the input and output formats we specify above so that we can run them, and your code must work on the SICE Linux computers.