B551 Assignment 3: Probability and Statistical Learning for NLP

Fall 2020
Due: Sunday November 29, 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, specifically related to Natural Language Processing.

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 one of the SICE Linux systems; 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 the SICE Linux servers; 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; we've already assigned you to a group (see details below) 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 on the assignment, except in unusual circumstances; we will collect feedback about how well your team functioned in order to detect these circumstances. The requirements for the assignment are the same regardless of team size, but we expect that teams with more people will submit answers that are more "polished" — e.g., better documented code, faster running times, more thorough answers to questions, etc.

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

We've assigned you to a team; find it as usual by logging into IU Github, look for a repo called *userid1*-a3, *userid1-userid2*-a3, or *userid1-userid2-userid3*-a3, where the other user ID(s) correspond to your teammate(s). Now that you know their userid(s), you can write them an email at userid@iu.edu. To get started, clone the github repository using one of the two commands:

```
git clone git@github.iu.edu:cs-b551-fa2020/your-repo-name-a3
git clone https://github.iu.edu/cs-b551-fa2020/your-repo-name-a3
```

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 speech (e.g., "blueberries" is a noun but can't be a verb).

But in general, one has to look at all the words in a sentence to figure out the part of speech of any individual word. For example, consider the — actual! — 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

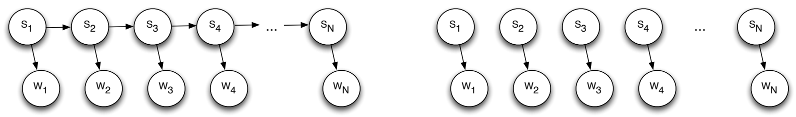

Figure 1: Bayes Nets for part of speech tagging: (a) HMM, and (b) simplified model.

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. While Viterbi gives the single best solution, it gives no notion of the *confidence* of its answer. One way of measuring this, on a per-word basis, is to compute the marginal probability of the part-of-speech found Viterbi for each individual word, given the entire sentence:

$$P(S_i = s_i^*|W)$$

   Intuitively, if we are confident of the part-of-speech estimated by Viterbit for a particular word, this number should be high, and otherwise it will be low.

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 two approaches (Simple and HMM), 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 two 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:

```
[djcran@raichu djc-sol]$ python3 ./label.py training_file testing_file
Learning model...
Loading test data...
Testing classifiers...
                      Simple     HMM  Magnus     ab integro seclorum nascitur  ordo .
  0. Ground truth     -48.52  -64.33    noun   verb     adv     conj     noun  noun .
    1. Simplified     -47.29  -66.74    noun   noun    noun      adv     verb  noun .
           2. HMM     -47.48  -63.83    noun   verb     adj     conj     noun  verb .
    3. Confidence                      0.999  0.513   0.839    0.392    0.999 0.999 .

==> 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%
```

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: Code breaking

You've intercepted a secret message that is encrypted using both of two techniques. In **Replacement**, each letter of the alphabet is replaced with another letter of the alphabet. (For example, all `a`'s may have been replaced with `f`'s, `b`'s with `z`'s, etc.) Unfortunately, we don't know the mapping that was used. In **Rearrangement**, the order of the characters is scrambled. For each consecutive sequence of n characters, the characters are reordered according to a function that maps character indices to character indices. For example, if $n = 4$ and the mapping function is $f(0) = 2, f(1) = 0, f(2) = 1, f(3) = 3$, then the string `test` would be rearranged to be `estt` (because the character at index 0 was moved to index 2, index 1 was moved to index 0, etc). We don't know the mapping function that the encoder used, but we do know that $n = 4$.

How can we decrypt a document without knowing the encryption tables? Probabilistic methods come to the rescue. For any given sequence of characters, we can score how "English-like" it is by viewing language as simple a Markov chain over letters of the alphabet. We can define the probability that a document D was generated from the English language, $P(D) = \prod_i P(W_i)$, where W_i is the i-th word of the document, and

$$P(W_i) = P(W_i^0) \prod_{j=1}^{|W_i|-1} P(W_i^{j+1}|W_i^j),$$

where $P(W_i^j)$ refers to the j-th letter of word i. Now let's say we randomly applied different decryption tables to an encrypted document. For each of those candidate decryptions D, we can calculate $P(D)$, and then choose the one that is highest — the one that maximizes the likelihood of the data.

Unfortunately, trying all possible tables is impossible because the number of possible codes is unthinkably enormous — there are 26! possible replacement code books and 4! possible rearrangement codes, for a total of about 10 trillion quadrillion combinations. Here's an alternative, based on something called the Metropolis-Hastings algorithm:

1. Start with a guess about the encryption tables. Call the guess T.
2. Modify T to produce a new guess, T'. The modification could be switching two letters in one of the tables, for example.
3. Decrypt the encoded document using T to produce document D, and decrypt the document using T' to produce D'.
4. If $P(D') > P(D)$, then replace T with T'. Otherwise, with probability $\frac{P(D')}{P(D)}$, replace T with T'.
5. Go to step 2.

Write a program to break codes of the above type. Your program should be run like this:

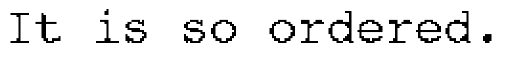
It is so ordered.

Figure 2: Our goal is to extract text from a noisy scanned image of a document.

```
python3 ./break_code.py encoded_document english_corpus output
```

where *encoded_document* is the name of a encrypted file, *english_corpus* is a document containing some English text, and *output* is the file in which to write the final decrypted output. The purpose of the English corpus is to estimate the probabilities needed to compute the probabilities above. (Intuitively, the idea is that your program is searching for a code such that, when used to decrypt the document, the statistics of the decrypted document match the statistics of known English text.)

Your code should output the best possible decryption it can find within a time limit of about 10 minutes. You might want to consider variations on the above algorithm to improve performance, such as running the algorithm multiple times and using the best (highest-probability) result. Make sure to explain these design decisions in your report. For simplicity, you can assume that the input and output documents consist only of lowercase letters and spaces — no punctuation or capital letters.

Part 3: 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 Parts 1 and 2 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, just like in Part 2.

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 Parts 1 or 2 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:

```
[djcran@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 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 $(100 - m)\%$.

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.