B551 Assignment 3: Probability and Statistical Learning for NLP

Fall 2019
Due: Sunday Dec 1, 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

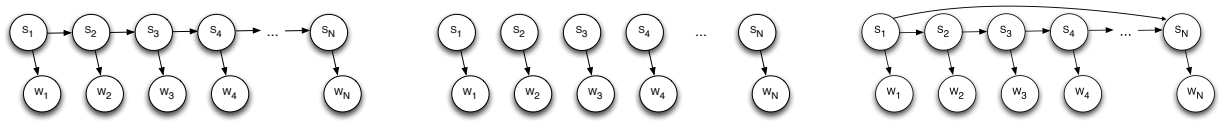
Coding requirements. For fairness and efficiency, we use a semi-automatic program to grade your submissions. As usual, we require that: 1. You must code this assignment in Python 3; 2. Make sure to include a #! line at the top of your code; 3. You should test your code on one of the SICE Linux systems; 4. Your code must obey the input and output specifications given below. 5. 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 6. 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.



(a) (b) (c)

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

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-fa2019/your-repo-name-a3
git clone https://github.iu.edu/cs-b551-fa2019/your-repo-name-a3
```

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. A 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, 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

[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.

(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(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 longer-range dependencies between words. But it's not an HMM, so we can't use Viterbi. Implement Gibb's 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, 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 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`.

[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.

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:

```
./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: Spam classification

Let's consider a straightforward document classification problem: deciding whether or not an e-mail is spam. We'll use a bag-of-words model, which means that we'll represent a document in terms of just an unordered "bag" of words instead of modeling anything about the grammatical structure of the document. If, for example, there are 100,000 words in the English language, then a document can be represented as a 100,000-dimensional vector, where the entries in the vector corresponds to a binary value — 1 if the word appears in the document and 0 otherwise. Of course, most vectors will be sparse (most entries are zero).

Implement a Naive Bayes classifier for this problem. For a given document D, we'll need to evaluate $P(S = 1|w_1, w_2, ..., w_n)$, the posterior probability that a document is spam given the features (words) in that document. Make the Naive Bayes assumption, which says that for any $i \neq j$, w_i is independent from w_j given S. (It may be more convenient to evaluate the likelihood (or "odds") ratio of $\frac{P(S=1|w_1,...w_n)}{P(S=0|w_1,...,w_n)}$, and compare that to a threshold to decide if a document is spam or non-spam.)

To help you get started, we've provided a dataset in your repo of known spam and known non-spam emails, split into a training set and a testing set. Your program should accept command line arguments like this:

```
./spam.py training-directory testing-directory output-file
```

The *training-directory* can be assumed to contain two subdirectories called `spam` and `notspam`, containing email files that can be used to estimate the needed probabilities of the model. The *testing-directory* contains test emails, one per file; your program should output a `output-file` in a format like this:

```
00393.85c9cd10122736d443e69db6fce3ad3f spam
01064.50715ffeb13446500895836b77fcee09 notspam
```

and so on, where the first part of each line is a filename and the second is predicted class (spam or notspam).

We have not prepared skeleton code this time, so you may prepare your source code however you'd like.

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.