Assignment 3: Adversarial Search and Probabilistic Inference

B551, Elements of Artificial Intelligence, 2024

Due: 11:59pm, Friday, November 22

This assignment will give you experience with adversarial search for game playing and the chance to apply probabilistic inference methods to real-world problems in natural language processing.

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 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 IU GitHub. All the people on the team will receive the same grade, except in unusual 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.

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) the contribution of each team member.

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.

1   Part 0: Getting started

For this project, you can find your starter code by logging into IU Github at https://github.iu.edu/cs-b551-fall2024 and navigating to the repository a3-release. The final team names will be published in the following format: userid1-a3, userid1-userid2-a3, or userid1-userid2-userid3-a3, 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 teammates, you can write them at userid@iu.edu.

To get started, clone the github repository:

git clone git@github.iu.edu:cs-b551-fall2024/a3-release.git

If that doesn't work, instead try:

git clone https://github.iu.edu/cs-b551-fall2024/a3-release.git

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

Part 1: Tic-Tac-what?

Cam and Adi, graduate students in computer science, spent their free time between assignments playing tic-tac-toe. Frustrated that Adi always insisted on playing first and knew the optimal strategy (leading to inevitable draws), Cam sought to create a variant of the game where he might have a chance to win.

The rules of this variant are as follows:

1. The game is played on a grid board

2. Unlike regular Tic-Tac-Toe, both players use X's (there are no O's)

3. Players take turns placing a single X in any empty cell and moves cannot be skipped or passed

4. Players must place exactly one X on their turn and a cell that already contains an X cannot be used again

5. The player who completes a line of specified length of X's loses the game (the line can be vertical, horizontal or diagonal)

In this part of the assignment, build a function to play the best possible move for a given board configuration. The empty positions in the board are denoted by '.' and the occupied positions are denoted by 'x'. Your program should take the input as a string representing the board (`board_string`), number of rows (`n`), number of columns (`m`) and length of line (`length`):

```
sahmaini@silo:$ python3 solver.py board_string n m length
```

For example:

```
sahmaini@silo:$ python3 solver.py 'xx..........' 3 4 3

Starting from initial board:
xx..
....
....

Deciding the next step...

Here's what we found:
xx..
..x.
....
```

We've provided you with some skeleton code to get you started. You should modify the solver function in solver.py file to return the new board state. We have also provided some inital test cases in test_part1.py. To test the code -

```
sahmaini@silo:$ python3 -m pytest -v
```

We encourage you to develop and run additional test cases to thoroughly validate your implementation.

Part 2: Part-of-speech tagging

A basic problem 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.). This is valuable for improving the performance of NLP systems for tasks such as machine translation.

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 and 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 —- 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. 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.)
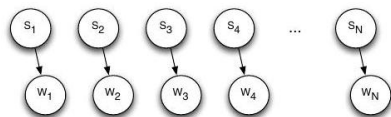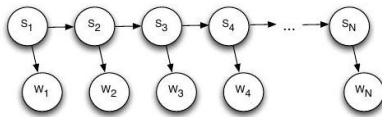

Figure 1: Simplified Model   Figure 2: HMM

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),

---

[1] If you didn't know the term "adposition", neither did we. 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.

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,

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

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

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

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

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 1-2 on each sentence in the testing file. For the result generated by both approaches (Simple and HMM), 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 models in Figure 1 and 2. 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    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   .

 ==> 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 %
```

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.

What to turn in

Be sure to include a report, as described at the start of this document. Create a private repository using the team name as described in Part 0, 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 two 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

---

[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.

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.