B551 Assignment 2: Games and Bayes

Fall 2018
Due: Wednesday October 24, 11:59PM
(You may submit up to 48 hours late for a 10% penalty.)

This assignment will give you practice with adversarial games and with Bayesian classifiers.

You'll work in a group of 1-3 people for this assignment; we've already assigned you to a group (see details below) based on the input you provided with your A1 team feedback. (We tried to accommodate as many of your requests as possible, but we could not satisfy all of them, in part because some people made requests that conflicted with those of others.) You should only submit **one** copy of the assignment for your team, through GitHub, as described below. All 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. Please read the instructions below carefully; we cannot accept any submissions that do not follow the instructions given here. Most importantly: please **start early,** and ask questions on Piazza or in office hours.

The following problems require you to write programs in Python. You may import standard Python modules for routines not related to AI, such as basic sorting algorithms and basic data structures like queues. You must write all of the rest of the code yourself. If you have any questions about this policy, please ask us. We recommend using the CS Linux machines (e.g. `burrow.soic.indiana.edu`). You may use another development platform (e.g. Windows), but make sure that your code works on the CS Linux machines before submission. We will release the output format testing code before the deadline.

For each programming problem, please include a detailed comments section that explains your approach, the assumptions and/or design decisions you made, any problems you faced, etc. (You can think of this as your opportunity to argue for why you deserve a good grade. For example, maybe you experimented with several other approaches that didn't work out; here's where you can tell us about them). Also, include the answers to any questions asked below in the assignment.

Academic integrity. You and your teammates 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). However, the work and code that you and your partners submit must be your group's own work, which your group personally designed and wrote. You may not share written answers or code with any other students except your own partner, nor may you possess code written by another student who is not your partner, either in whole or in part, regardless of format.

Part 0: Getting started

You can find your assigned teammate by logging into IU GitHub, at `http://github.iu.edu/`. In the upper left hand corner of the screen, you should see a pull-down menu. Select `cs-b551-fa2018`. Then in the yellow box to the right, you should see a repository called *userid1-userid2-userid3*-a2, where the other user IDs correspond to your teammate(s).

To get started, clone the github repository:

`git clone git@github.iu.edu:cs-b551-fa2018/`*`your-repo-name`*`-a2.git`

where *your-repo-name* is the one you found on the GitHub website above.

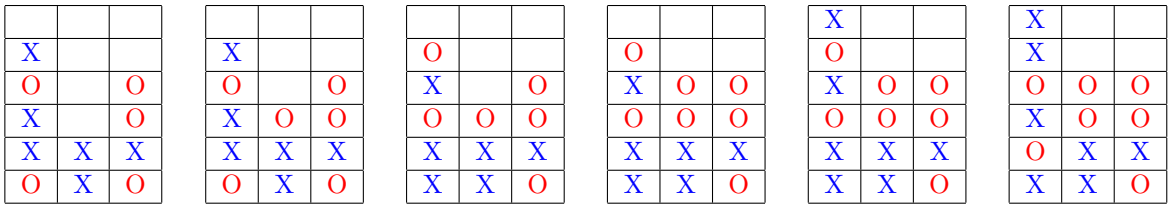

Figure 1: A sample series of moves at the end of a game of Betsy with $n = 3$. From the initial board in (a), red chooses to drop a pebble in the second column, resulting in (b). Blue then rotates the first column to give (c). Red drops another pebble in the second column to give (d). Blue drops a pebble in the first column to give (e). Finally, red rotates the first column, and wins the game by completing a row within the top n rows of the board.

Part 1: Betsy

A popular game in a certain exotic, far-off land[1] is called *Betsy*. It's played on a vertical board that is n squares wide and $n + 3$ squares tall (where n is often 5 for beginners, but can grow quite large in the professional tournaments). The board starts off empty, with each of the two players (red and blue) given $\frac{1}{2}n \times (n + 3)$ pebbles of their own color. Blue goes first, choosing one of two possible types of moves.

- *Drop:* Choose one of the n columns, and drop a blue pebble into that column. The pebble falls to occupy the bottom-most empty square in that column. The player is not allowed to choose a column that is already full (i.e., already has $n + 3$ pebbles in it).
- *Rotate:* Choose one of the n columns, remove the pebble from the bottom of that column (whether red or blue) so that all pebbles fall down one square, and then drop that same pebble into the top of that column. The player is not allowed to choose an empty column for this type of move.

After making a move, blue checks the top n rows of the board to see if they have completed a row of n blue pebbles, a column of n blue pebbles, or one of the two diagonals of blue pebbles. The bottom three rows of the board are ignored during this check. If a row, column, or diagonal has been completed in blue, blue wins! Otherwise, red makes the same check and wins if any row, column, or diagonal has been completed with red. Note this means that if blue completes a row, column, or diagonal of blue pebbles, they win *even if* they have also completed a row, column, or diagonal of red. If no one has won, player red takes their turn, either dropping a red pebble into an incomplete column or rotating a non-empty column. Figure 1 shows several sample moves from a game in progress, with $n = 3$.

Your task is to write a Python program that plays Betsy well. Use the minimax algorithm with alpha-beta search and a suitable heuristic evaluation function.

Your program should accept a command line argument that gives the value of n, and the current state of the board as a string of $n \times (n + 3)$ characters (from top to bottom and left to right, i.e. in row-major order), each of which is one of: `.` for an empty square, `x` for blue pebble, and `o` for a red pebble. For example, the encoding of the board in Figure 1(a) would be:

```
...x..o.ox.oxxxoxo
```

More precisely, your program will be called with three command line parameters: (1) the value of `n`, (2) the current player (`x` or `o`), (3) the state of the board, encoded as above, and (4) a time limit in seconds. Your

[1] Okay, it's really just Central Pennsylvania. It's not my fault I was born there!

program should then decide a recommended single move for the given player from the given current board state, and display the recommend move and the new state of the board after making that move, *within the number of seconds specified,* in a format like this:

```
move new_board
```

where `move` is either a positive number indicating a column (ranging from 1 to n) in which to drop a pebble, or a negative number indicating a column to rotate (e.g., -3 means to rotate column 3). Displaying multiple lines of output is fine as long as the last line is the recommended move and board state.

For example, two runs of your program (corresponding to the first two moves of Fig. 1) might look like:[2]

```
[djcran@macbook]$ ./betsy.py 3 o ...x..o.ox.oxxxo.o 5
Shhh... I'm thinking!
Hey, in the time it takes you to read this sentence, I'll have considered
5 billion board positions. But it's cute that you're still trying to beat me...

I'd recommend dropping a pebble in column 2.
2 ...x..o.oxooxxxoxo
[djcran@macbook]$ ./betsy.py 3 x ...x..o.oxooxxxoxo 5
Shhh... I'm thinking!
Sure, you could unplug me, but within 500ms I can command every computer on Earth
to delete any trace that you ever existed. You're welcome.

I'd recommend rotating column 1
-1 ...o..x.ooooxxxxxo
```

In your source code comments, explain your heuristic function and how you arrived at it, and any other interesting approaches you tried, problems you faced, etc.

The tournament. To make things more interesting, we will hold a competition among all submitted solutions. We will not reveal ahead of time the time limit or board size, but we plan to hold multiple tournaments with different values of each. While the majority of your grade will be on correctness, programming style, etc., a small portion may be based on how well your code performs in the tournaments, with particularly well-performing programs eligible for prizes including extra credit points.

Notes and hints: Your code must conform to the interface standards mentioned above! The last line of the output *must* be the move and new board in the format given, without any extra characters or empty lines. We will provide an output checker to help you verify this. We will also provide a program that will allow you to play against other teams in the class without having to share your Python code (which would violate the academic integrity policies of the course).

Note also that your program cannot assume that the game will be run in sequence from start to end; given a current board position on the command line, your code must find a recommended next best move. Your program can write files to disk to preserve state between runs, but should correctly handle the case when a new board state is presented to your program that is unrelated to the last state it saw.

Our test program will enforce the time limit by killing your program after the time limit is exceeded. Since we will only look at the last line that your program produces, an easy way of dealing with the time limit is to quickly calculate and print a suggested "rough-draft" move, and then print out better moves as it finds them. Our test program will kill your program after the time limit and only look at the best move it has displayed, which presumably is the best it know about so far.

[2] "Trash talk" is optional. :)

Part 2: Tweet classification

A classic application of Bayes Law is in document classification. Let's examine one particular classification problem: estimating where a Twitter "tweet" was sent, based only on the content of the tweet itself. We'll use a bag-of-words model, which means that we'll represent a tweet in terms of just an unordered "bag" of words instead of modeling anything about its grammatical structure. In other words, a tweet can be modeled as simply a histogram over the words of the English language (or, more generally, all possible tokens that occur on Twitter). If, for example, there are 100,000 words in the English language, then a tweet can be represented as a 100,000-dimensional binary vector, wherein each dimension there is a 1 if the word appears in the tweet and a zero otherwise. Of course, vectors will be very sparse (most entries are zero).

Implement a Naive Bayes classifier for this problem. For a given tweet D, we'll need to evaluate $P(L = l|w_1, w_2, ..., w_n)$, the posterior probability that a tweet was taken at one particular location (e.g., l = Chicago) given the words in that tweet. Make the Naive Bayes assumption, which says that for any $i \neq j$, w_i is independent from w_j given L.

To help you get started, we've provided a dataset in your GitHub repo of tweets, labeled with their actual geographic locations, split into a training set and a testing set. We've restricted to a set of a dozen North American cities (Chicago, Philadelphia, etc.), so your task is to classify each tweet into one of twelve different categories. Train your model on the training data and measure performance on the testing data in terms of accuracy (percentage of documents correctly classified).

Your program should accept command line arguments like this:

```
./geolocate.py training-file testing-file output-file
```

The program should then load in the training file, estimate the needed probabilities to build a Bayesian model, and apply them to each tweet in the testing file, and then write the results into output-file. The file format of the training and testing files is simple: one tweet per line, with the first word of the line indicating the actual location. Output-file should have the same format, except that the first word of each line should be your estimated label, the second word should be the actual label, and the rest of the line should be the tweet itself. Your program should also output (to the screen) the top 5 words associated with each of the 12 locations (i.e. the words for which $P(L = l|w)$ is the highest for each l).

The goal is to get as high an accuracy as possible in testing, including on the separate test dataset we'll use to test your code. You'll have to make various design decisions in doing this, e.g. whether to use all "words" (i.e. Twitter tokens) or just the most common ones, whether to keep punctuation or remove it, whether to keep capitalization or remove it, etc. *Please describe these design decisions and any experimentation you use to arrive at them in your report.*

Hints: Don't worry, at least at first, about whether the "words" in your model are actually words. Just treat every unique space-delimited token you encounter as a "word," even if it's misspelled, a number, a punctuation mark, etc. It may be helpful to ignore tokens that do not occur more than a handful of times, however. To perform classification, you'll need to compute the posterior probability for each of the 12 cities and then choose the maximum. Note that this means you don't have to actually compute the prior on words, i.e. the denominator of Bayes Law, since it is the same across all 12 categories and is always positive (so that maximizing the numerator is the same as maximizing the full posterior).

Extra credit[3]

In a momentary lapse in common sense, an otherwise mild-mannered professor of computer science made the arrogant claim that he "could implement a program to play perfect Chess in 20 lines of Python code."[4] The point wasn't to brag about his coding abilities, but simply to underscore the point that in theory, game-playing AI can be easily implemented; the hard part is implementing it in a way that avoids simply searching the space of all possible games. Unfortunately, several persistent students called his bluff, leading to some frantic late-night, jet lag-assisted coding to prove to himself he could do it. His (theoretically) perfect chess program takes 17 lines of code, and a more practical version with a heuristic function takes 19. Can you do better? The rules are: (1) you're not allowed to import any modules, (2) blank lines do not count, but multi-line statements count for multiple lines, (3) must use reasonably meaningful variable and function names, (4) no line may have more than 225 characters, (5) I/O, test code, etc. doesn't count, (6) assume simplified Chess rules that do not include en passant or castling, and players may move into Check (but then will lose the next turn when their King is taken), and (7) no external data or program files may be loaded or used.

Here's what a run of the program should look like,

```
[djcran@macbook]$ ./chess.py w RNBQKBNRPPPPPPPP..............................pppppppprnbqkbnr 10
Thinking! Please wait...

Hmm, I'd recommend moving the Pawn at row 2 column 3 to row 4 column 3.
New board:
RNBQKBNRPP.PPPPP..........P....................pppppppprnbqkbnr
```

where the command line arguments are similar as with Betsy: w means the white player, 10 is a 10 second time limit, and the board state is a 64-character string with R, N, B, Q, K, and P representing rook, knight, bishop, queen, king, and pawn, respectively, and upper case pieces are white and lower case are black.

What to turn in

Turn in the files required above by simply putting the finished version (of the code with comments and PDF file for the first question) 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.

[3] Don't attempt unless you're completely happy with your submission for the rest of the assignment. This problem is mostly for fun. We want you to learn how to write clear, concise, easy-to-understand, easy-to-maintain code, whereas this problem encourages writing code that is hard to maintain and difficult to understand. But still, it may be fun :-).

[4] In his defense, he had just returned from China and hadn't slept in 36 hours.