Stats 102A

Midterm Project Instructions

July 11, 2024

General Notes

- You will submit a minimum of three files, the core files must conform to the following naming conventions (including capitalization and underscores). 123456789 is a placeholder, please replace these nine digits with your nine-digit UID. The files you must submit are:

  1. `123456789_stats102a_midterm.R` An R script file containing all of the functions you wrote for the homework. The first line of your .Rmd file, after loading libraries, should be sourcing this script.
  2. `123456789_stats102a_midterm.Rmd` Your markdown file which generates the output file of your submission.
  3. `123456789_stats102a_midterm.html/pdf` Your output file, *either* a PDF or an HTML file depending on the output you choose to generate.
  4. `included image files` You may name these what you choose, but you must include all image files you generated otherwise your file will not knit.

  If you fail to submit any of the required core files you will receive **ZERO** points for the assignment.

  If you submit any files which do not conform to the specified naming convention, you will receive (at most) **half credit** for the assignment.

- Your coding should adhere to the tidyverse style guide: https://style.tidyverse.org/ <!–
- All flowcharts should be done on separate sheets of paper, but be included, inline as images, in your final markdown document. –>
- Any functions you write should be included in a separate functions file.
- Your .Rmd file must knit. If your .Rmd file does not knit you will receive (at most) **half credit** for the assignment.

The two most common reasons files fail to knit are because of workspace/directory structure issues and because of missing include files. To remedy the first, ensure all of the file paths in your document are relative paths pointing at the current working directory. To remedy the second, simply make sure you upload any and all files you source or include in your .Rmd file.

NOTE: *Everything* you need to do this assignment is here, in your class notes, or was covered in discussion or lecture.That being said, this is a big assignment, and you will need to get help from me, your TA, your class mates, the internet, and LLMs.

- **DO** look for solution strategies online.
- **DO** use chatGPT, codepilot, or other large language models or AI helpers.
- **YOU MAY** collaborate on this assignment.
  - Each person **MUST** create **THEIR OWN** submission of this assignment in its entirety. You are solely responsible for your submission, this is not a group project.

Battleship Specification

Rules

From Wikipedia:

Traditionally, the game is played on four grids, two for each player. The grids are typically square, usually 10×10, and the individual squares in the grid are identified by letter and number. On a primary grid the player arranges their own ships and keeps a record of the shots by the opponent. On the secondary grid the player records their own shots as hits or misses.

Before play begins, each player secretly arranges their ships on their primary grid. Each ship occupies a number of consecutive squares on the grid, arranged either horizontally or vertically [but not diagonally]. The number of squares for each ship is determined by the type of the ship. The ships cannot overlap (i.e., only one ship can occupy any given square in the grid) [but they can touch]. The types and numbers of ships allowed are the same for each player. These may vary depending on the rules.

The default ships we will use come from the 2002 Hasbro edition of the game

Ship Number	Name	Size
1	Aircraft Carrier	5
2	Battleship	4
3	Destroyer	3
4	Submarine	3
5	Patrol Boat	2

After the ships have been positioned, the game proceeds in a series of rounds. In each round, each player takes a turn to announce a target square (e.g. “c-7”) in the opponent’s grid which is to be shot at. The opponent announces whether or not the square is occupied by a ship, and if it is a “miss”, the opponent player marks their primary grid with a white peg; if a “hit” they mark this on their own primary grid with a red peg. The attacking player notes the hit or miss on their own “tracking” grid with the appropriate color peg (red for “hit”, white for “miss”), in order to build up a picture of the opponent’s fleet.

When all of the squares of a ship have been hit, the ship’s owner announce the sinking of either the Carrier, the Cruiser, the Submarine, the Destroyer, or the titular Battleship. If all of a player’s ships have been sunk, the game is over and their opponent wins.

Our game will be a variation where the sinking of a ship ***is not*** announced. Make sure you play a few rounds of battleship online. Here is an online game with similar rules: https://www.cbc.ca/kids/games/play/battleship

(1) Create several classes of objects.

These are just descriptions of what an object of this class should look like, you do not have to write any code for part (1) of this assignment. You will use the S3 system to create objects which does not rely on formal class definitions. Instead, you will create constructor functions for these objects in part (2). Make sure that your constructor functions in part (2) output an object of the correct class (usually a list with class attribute set to the class name) that contains all the necessary elements.

`battleship`: Essentially, this is the game object. Everything about the current status of the game is included here.

`fleets` This is a length-two list of the players' fleets.

`history` This is a tibble object with four columns:

from A character vector of the admiral firing the shot.

to A character vector of the admiral who is being fired upon.

target A character vector of the target of the shot. e.g. "F-9", "B-6", etc.

hit A logical vector of whether the shot hit a ship.

`fleet`: An object which represents one admiral's fleet.

`admiral` A character string representing the player in charge of the fleet.

`ocean` A length-two numeric vector specifying the size and shape of the ocean the fleet occupies. Be aware that the first entry in ocean specifies the number of rows and the second entry specifies the number of columns. The Hasbro 2002 rules specify that the board is 10x10 squares with the rows labeled with capital letters A-J starting with A on top, and the columns labeled 1-10 with 1 on the left. This means that `ocean = c(5, 7)` specifies an ocean with 5 rows (lettered A-E) and 7 columns (lettered 1-7), with valid target squares going from "A-1", "A-2", "A-3", ..., "E-6", "E-7".

`ships` A list of ship objects which are members of the fleet.

`ship`: An object representing one ship.

`name` A character string. The name of the ship.

`size` An integer. The number of spaces the ship occupies.

`position` A length-two character vector of the bow (front) and stern (back) position of the ship, e.g., `position = c("A-1", "A-6")`. Positions are of the format "upper case letter-number" without any spaces.

`hits` A length(ship$size) logical vector indicating which portions of the ship have been hit.

`sunk` Logical, indicating if the ship is sunk.

(2) Create functions:

Class Constructors: These constructor functions are how you will create objects of a certain class. Make sure that your constructors return an object (usually a list) with the class attribute set to the respective class name that contains all the elements listed in part (1). Whenever you will create an object of certain class (for example ship) you should do so by calling your constructor function (for example ship()).

`ship()` Creates one `ship` object

`name` A character string, the name of the ship, e.g. "Battleship," "Submarine," etc.

`size` An integer, the number of spaces the ship occupies.

`fleet()` Creates one `fleet` object.

`admiral` A character string to represent the Admiral of the fleet.

`ocean` *Optional.* Default: `ocean = c(10, 10)` A length-two numeric vector representing the dimensions of the ocean the fleet will occupy. For the sake of this project you may assume ocean dimensions will be *at least* 5×5 and *at most* 25×25

`ships` *Optional.* Default: `ships = NULL` A variable-length list object containing one or more `ship`-class objects.

If `ships = NULL`, the result of `default_ships()` should be used.

`battleship()` Creates a game object.

`fleets` *Optional.* Default: `fleets = list()`. A list of `fleet` objects in the game.

If `length(fleets) < 2` make either 1 or 2 default fleets (standard ships on a standard board). If a default `fleet` is generated the admiral should be "Player 1" or "Player 2" as appropriate.

Gameplay

`play_bs()` The workhorse function for playing a single game of Battleship. This function sets up the game and calls on each of the two players to make a move over and over again until there is a winner. It has the following arguments:

`players` Who is playing the game, this should be the names of functions which return a target string like "E-6". The default should be for a one-player game `players = c("human", "ai_123456789")`. The first player in `players` always goes first.

`strengths` The strength of the players. This is a character vector of length two, specifying the strength of the players. The default should be `strengths = c(9, 9)`. If the first player is human the first entry in strengths is unused.

`verbose` logical. Should a turn counter, each player's actions and whether or not a ship was hit be printed out to the console.

`plot_before_turn` Should the game be plotted before a player's turn. This a character of length 1, with default `plot = "none"` and options `"player 1", "player 2", "both"` stating whether the game should be plotted before a players turn. make sure you only plot information known to that player. This should be handed over to the `which` argument of your `plot.battleship()` function

The return value for this function is a list object. The minimal contents of this list should be `winner =` the name of the admiral who won the game. You may need to include more data in your return object to answer all of the questions in this project.

`human()` The function which polls a human player for a target. Use the `readline` function to get an input. It should accept a character string indicating a row (an appropriate letter) and a column (an appropriate number). Accepted responses should be in the form of "F-3", "J-10", etc. (**Hint:** The `substr()` or the `strsplit()` functions may be helpful for processing targets.) If a human inputs an unacceptable string like "A5", "5-5", etc. ask for another input with a warning; do not crash the entire game with an error message. Allow lower case letters as human input but convert them to uppercase letters before returning, e.g., the input "f-2" should be returned as "F-3".

`ai_123456789()` This is your bot, make sure you change its name to `ai_yourStudentIdNumber()`. It should take *either* the following three arguments:

`battleship` The current game object. **Note:** Though your bot has access to the *entire* game object which notably includes your opponent's ship placements, you are to limit your bot's access to the following items:

`history` The object detailing the shots and hits in the game.

`ocean` The objects detailing the size and shape of your opponents `ocean`, so you know the bounds of the regions you need to look for ships in.

`size` The vector of ship sizes so you know how many ships of each size you are looking for.

Any bots which are found to access the opponent's ship placements or status (`position`, `hits`, or `sunk`) will be excluded from the tournament section and receive a 0 for that portion as well as a 0 for all portions of this project involving the AI agent you were to write. **DON'T BE A CHEATER!**

`strength` *Optional.* Default: `strength = 9`. This should be the level of play of your bot. The default should be `strength = 9`, this is the best bot you have made at the time of your submission. You should also have a `strength = 0` bot which plays completely randomly, with no strategy, but never in the same spot twice. It **is not necessary** to have more than these two levels, but it is required to have at least these two. You may want to explore and experiment with different strength bots for a variety of reasons, but you are not required to.

`memory` *Optional.* Default: `memory = list()`. This is an argument that allows you to store any information, computed quantities, etc. from one turn to the next. This can help make your code run a lot neater, faster, and more efficiently. This function will return a memory as well as a target location. This memory is fed back to your bot through the memory argument in it's next turn. This can help you write overarching multi-turn strategies without recomputing them in every turn.

No bot shall ever fire upon the same spot twice nor shall any bot fire on a spot outside the confines of the designated ocean.

The return value of this function should be a list of length two, with the first entry being a character string named `target` representing a target square in the opponent's ocean grid like "D-6", "I-7", etc., and the second entry is a list named `memory`. This memory will be handed over to your AI in it's next turn.

Make sure that this function is entirely self-contained. It cannot access any functions or objects outside of itself. It knows and can use only what is handed over to it in its three arguments.

or the function should take one argument:

`fleet` *Optional.* No default. If a `fleet` object is passed to your AI function it should return the same `fleet` object with updated `position` values for the `ship` objects in the fleet. That is, it places the ships.

`default_ships()` Create a list object containing the five default ships (see rules), with no assigned positions.

`position_fleet()` A function to assign ships a position in the ocean.

`fleet` A fleet object.

`positions` *Optional.* Default: `positions = NULL`. A list of the same length as the ship list for the `fleet`. Each list item shall comprise a length two character vector indicating the start and end position of each ship in the ship list.

If no `positions` list is provided, the ships shall be randomly placed in the ocean. The return value for this function should be a fleet object with updated ship positions.

(3) Create methods

For the following functions create methods for the specified classes. Unless specified, you are free to implement these however you like, but you should think carefully about what makes sense to do, that is, for example, what should printing a `ship` entail? Or a `fleet`? How would you summarize a game (a `battleship` object)?

- `print()`

`ship` Print a meaningful representation of a `ship` object, something other than simply dumping the contents with the default printing method.

`fleet` Print a meaningful representation of a `fleet` object, something other than simply dumping the contents with the default printing method.

`battleship` Print a meaningful representation of a `battleship` object, something other than simply dumping the contents with the default printing method.

- `plot()`

`fleet` This should produce a graphical representation of one player's game board, with their ships placed appropriately, including indications where the ship has been hit.

`battleship` This should produce a graphical representation of the current state of the game. It has an additional argument `which` stating whose information should be plotted. This a character of length 1, with default `which = "both"` and options `"player 1", "player 2"` stating whether only information known to that player should be plotted. If `which = "both"`, both players' oceans should be drawn with each player's ships and their opponent's hits and misses. If textttwhich = "player 1", the ocean's of both players should be shown, and the ocean of the first player should show the ships of the first player and the hits and misses of the second player. The ocean of the second player should only show the hits and misses of the first player, but not the ships of the second player. If textttwhich = "player 2", follow the same logic with inverse roles.

- `summary()`

`fleet` Summarize a `fleet` object in some meaningful way.

`battleship` Summarize a `battleship` object in some meaningful way.

(4) Simulate 100,000 games for each of the following conditions:

- `naive(strength = 0)` vs `naive(strength = 0)`
- `naive(strength = 0)` vs `smart(strength = 9)`
- `smart(strength = 9)` vs `naive(strength = 0)`
- `smart(strength = 9)` vs `smart(strength = 9)`

Questions

A) Showcase your game

Show that your functions work and that you have created a working version of battleship.

a) Create a ship called "Aircraft Carrier", a fleet with and without locations, and battleship game after 10 turns. Call your `print`, `plot`, and `summary` functions to each of them (as appropriate).

b) include a screenshot of you playing against your AI after 10 turns.

B) Standard Game Simulations

Answer the following questions for each of your simulations in (4):

a) What is the minimum number of turns the winning player needed to win the game? How does this compare to a theoretical minimum.

b) What is the maximum number of turns the winning player needed to win the game? How does this compare to a theoretical maximum.

c) What is the distribution of the number of unsunk ships the winning player had remaining?

d) What is the distribution of the number of hits the losing player made?

e) In what proportion of games has the winner lost their Patrol Boat?

f) In what proportion of games is the losing player's last ship the Patrol Boat?

Also answer the following questions:

g) Make a two-way relative frequency table of the proportion of times Player 1 (whoever goes fist) wins.

h) Test the hypothesis that order of play is not a statistically significant factor in determining who wins. Use a 5% significance level, and interpret the *p*-value.

i) Test the hypothesis that the type of AI player is not a statistically significant factor in determining who wins. Use a 5% significance level, and interpret the *p*-value.

C) Handicapped Games

If you wrote your functions as specified in a general enough way, it should be possible to conduct a game between two different fleets on two different oceans. Experiment with changing the setup of the game to give one player an advantage over another. Perhaps you simply give one player a 9×9 board to hide their ships in and you give the other player an 11×11 board to hide in. Or you trade one player's Patrol Boat for the other player's submarine.

Try *at least* **three different** handicaps in favor of the naive AI against your smart AI with the goal of making it a fair game between the two.

Run 100,000 simulations and provide some summary statistics for each set of simulations. Don't worry if it's not exactly fair, even with the handicap, but it should change your results in the right direction.

D) Extensions

a) Describe in as much detail as possible, what you would need to change to implement the Salvo! variant of the game. (See rules.)

b) Identify at least two (2) ways you could modify or extend the game, other than the Salvo! variant, and describe in as much detail as possible what you would need to change to implement each of the extensions you identified.

E) Tournament

Your Battleship bots will be pitted against each other in a series of random, but fair, games (random ocean size, random fleets, same for both sides). 10% of the points for the Midterm Project will be allocated to your ranking in this tournament. You will get points for having a working submission, no matter how poorly it performs. Make sure that your `ai_123456789()` function is entirely self-contained, you will not be able to access any helper functions during the tournament. I will only read your `ai_123456789()` from your R file, and then I will pit it against your classmates `ai_123456789()`, so make sure it works by itself.

For anonymity, please add an attribute (in your .R file) to your AI function called `alt` which is a character string containing an alternate name to report your function's results under.

```r
ai_123456789 = function(...) {
  # your code here
}
attributes(ai_123456789) = list(alt = "The Professor")
ai_123456789
```

```
## function (...)
## {
## }
## attr(,"alt")
## [1] "The Professor"
```