Certificate 1 Session 2 — Script

[slide 1]
Last session we studied the concept of search and how we can apply different search strategies to systematically and exhaustively enumerate all possible nodes in a graph (or states in the search space). While both in depth-first search and in breadth-first search we sometimes cannot differentiate between various reachable states and process them in a *random* order, the other steps of the algorithms are *deterministic*: we know exactly what to do, no matter which (type of) graph is given to us as an input. Deterministic, *rule-based* algorithms were at the origins of AI, but they paved the way to more complicated, fuzzy and flexible approaches that modern systems currently use.
In this session we are going to have a first look at how we can adapt the behaviour of an algorithm or a system based on *data*. That is, we will still follow a certain strategy, but we will adjust the actual steps we take, or decisions we make, based on what the real-world data can tell us. This is not yet *machine learning*, but the simple approach we are going to try tonight will give a taste of what *learning from data* can give us, and will lead us to practical machine learning, or ML for short, in the next session.

[slide 2]
As we discussed, the general principle of the game of Battleships is to systematically find the opponent's ships and to hit them cell by cell. We can think of the game as a sequence of alternating modes: (*1*) in the *hunt mode*, we are blindly searching for a piece of ship on the unopened part of the opponent's board; then, whenever a ship is hit, we switch (*2*) to the *target mode*, in which we exhaustively search through the surrounding cells in order to sink the ship. We have implemented a search strategy for the target mode, and it is now time to improve on the hunt mode.

[slide 3]
Can we do better than blindly and randomly guess the next ship? Can we make an *educated guess*? Are all unopened cells equally probable to contain a ship? Let's see!

[slide 4]
We will assume from now on that there are patterns according to which users allocate their ships on the board. For example, sometimes people like to stack all ships in one area of the board, or to space them apart occupying the corners, etc.
More so, we conjecture that when we know something about the board, i.e., we have several cells open with hits or misses, we can predict a bit better where the rest of the ships are located, based on those patterns. So, whenever we are in the hunt mode, the idea is to *predict* where a ship probably is instead of guessing randomly.

[slide 5]
In order to make such predictions, we need to have some data, e.g., how users were previously playing the game, or how they were allocating the ships etc. Then we can make new decisions based on previous actions.
Many users have already played Battleships on the aigaming.com platform. From the logs of those plays, we have the ship arrangements on the board which the users created at the beginning of each play. We have saved more than *60* thousand board arrangements in a text file and uploaded it on Github — a website many developers use to store their coding projects. You can download the file and also access it from your Python code using the link: https://github.com/smartypanty/AIgaming-course/blob/master/C1S2/boards.txt.
NB! in the same location you can also find a test board set *test.txt* containing only 3 boards, so that you can test your code on it without having to wait for the bog dataset to load.

[slide 6]
Let's look at the structure of the *boards.txt* file. You can do it by either (*1*) downloading the file following the link, or (*2*) inspecting it directly on the Github website, or (*3*) printing the contents of the `boards` variable with `print(boards)`. Every board occupies a separate line of the file. The board is represented as a *list of lists*, each inner list corresponding to a row in the board. All in all, each board is a list of *8* rows. Ships are represented by numbers, e.g., *'3', '3', '3'* is a horizontally-oriented ship of length *3,* and four *'1'*s located the same position of four different rows correspond to a ship that is oriented vertically.

[slide 7]
In order to use the boards dataset, first of all, we need to import a Python module that will help us work with web content. For that, in the beginning of your bot file (where all the other imports are located) type `import urllib.request`. This module has functions that can open a url of some web page and access the content of this page. To do so, we need the second line `url = urllib,request.urlopen("")`, and as a string parameter we specify the url of the *boards.txt* file. This line will create a Python object `url` that we can now process. From `url`, we access the content of the webpage by a command `s = url.read()`, and we further add a function `decode()` that converts `s` from a sequence of bytes into a string. What is left is to split our huge input string `s` into separate boards: `boardStrings = s.splitlines()` (another way to do it is to explicitly specify that we want to split the string by the newline symbols: `s.split("\n")`). One further step is to remove any empty lines at the end of the input file with the function `strip()` so that we do not have empty lists in `boardStrings`: `boardStrings = s.strip().splitlines()`.
The resulting object is a list of strings named `boardString`, where each string is formatted as a python list (*["", "2", "", …]*). We now need to parse them and create a corresponding list for each of them (`for b in boardStrings`). For that we use the function `eval()`. The object boards is a list of lists, which you can check by printing it: `print(boards)`. You can also double-check that all the boards are loaded by printing the size of the `boards` list: `print(len(boards))`, and the answer should be *67174*.
NB! All the code we are using today can be found on the github page, in file *boards-parsing-online.py*.

[slide 8]
Now we are ready to use the boards data! Let us locate the exact piece of code that we are going to work on. Same as in the previous session, the main function is `calculateMove()`: it calculates the next move and returns it as the `move` variable. We need the `else`-statement as this is where we calculate the moves when the ships are deployed.

[slide 9]
At each move (except for the first round when we arrange ships), we parse the state of the opponent's board from the `gameState` dictionary: `oppBoard = gameState['OppBoard']`. `'M'` stands for a missed shot, `'H'` stands for a hit. As an example, look at the partially-opened board on the picture and the corresponding output list of lists `'OppBoard'`. We are in the middle of the game and we have partial information about the opponent's board.

[slide 10]
Now we will search through the log file and look for those boards that coincide with the opponent's board on the open, known cells. For example, we need those boards that (a) contain some ships in cells *3* and *4* of the first row, and also in cell *3* of the last row and cell *7* of the seventh row, AND and the same time (b) they do not contain ships in cells *1, 3, 5* of the third row, in cell *5* of the fourth row etc. One such board from the log file, that complies with these constraints, is given in the example. We will call such boards *similar boards.*

[slide 11]
In order to find similar boards, we need to do two things: (*1*) we need to identify all cells in the opponent's board that are open so far and that we use when searching for the similar boards; and (*2*) we need to find those boards in our dataset `boards`.

We create am empty list: `openedCells = []`. Then we iterate through all the cells on a board using two `for`-loops over indices `i` and `j` that range from *0* to *7*. For every possible cell we check whether the content of that cell in the opponent's board (`oppBoard[i][j]`) is known, i.e., it is not empty (`!= ''`), in which case we add that cell to the list of opened cells: `openedCells.append([i,j])`. Note that we represent a cell as a list of two indices, first corresponding to the row number and the second corresponding to the column number: `[i,j]`.
Now that the relevant, opened cells are collected, we create another empty cell `similarBoards = []`, and we go through all the boards (`for board in boards:`) and through all opened cells (`for cell in openedCells:`), and we check them. If a cell is a miss on the opponent's board, but contains a ship on the given board (i.e., is it not empty), we exit the inner `for`-loop and move to the next board — this is the task of the keyword `break`. Similarly, is a cell is a hit on the opponent's board, but it is empty on the current board, we move to the next one.
If, however, all the cells were checked and the board is indeed a similar board, i.e., the inner `for`-loop has ended normally, we move to the `else` clause and add the board to the list of similar boards: `similarBoards.append(board)`.
Here we can see that Python has `else`-clauses not only for `if`-statements, as in other programming languages, but also for `for`-loops, where it is executed whenever the loop was completed and not interrupted with a break.

[slide 12]
What remains now is to find cells on similar boards that are more likely to contain a ship. For that, we take all the similar boards and check which other cells on those boards contain ships, obviously excluding from the analysis cells that are already known and open. The intuition is that the more boards that are similar to the opponent's board contain a ship in cell *X*, the more likely it is to contain a ship in the opponent's board. In pseudocode, the procedure is for every unopened cell on the opponent's board to count the number of similar boards that contain a ship in it (the *score* of the cell), and then to pick the cell with the highest score as our next move.

[slide 13]
Now let's translate it into Python. We create an empty dictionary `targetCells` that will contain as keys cells that contain ships on similar boards, and as values the cell scores, i.e., the number of similar boards with a ship in those cells. Then we loop through the similar boards (`for board in similarBoards:`) and through every possible cell (again using two `for`-loops over indices `i` and `j`), in case it is not yet opened (`if [i,j] not in openedCells:`), we check whether the given board has a ship in it (`if board[i][j] != ''`). If it does, we update the information about that cell in the `targetCells` dictionary.
As we cannot use lists as keys in a Python dictionary, since they are not hashable, we cannot use our list representation of cells *[row_index, column_index]*. Therefore, we are going to represent a cell as a string, following the format that the Battleships game uses for moves: *{'Row': row_letter, 'Column': column_number}*. As you may remember from the previous session, there is a convenient auxiliary function in the bot code that does exactly that for us: `translateMove(i,j)` takes the row and column indices and outputs a `move` object, which we then can represent as a string using the predefined `str()` Python function: `cell = str(translateMove(i,j))`. Finally, we update the `targetCells` dictionary. The logic is the following: if the dictionary does not yet have an entry for that cell, create it and set its value to *1*; otherwise update the value incrementing it by *1*. One can obviously implement this logic using an `if–else`-statement. However, Python provides a more elegant way of doing it: one can call a predefined function `get(key, value)` on a dictionary, that returns the currently stored value of the given `key`, if such value exists, and if it doesn't, the function stores the specified `value` for the given `key` in the dictionary. One can see it as setting default values to keys. What we can do is to set the default value to *0* and always increment it by *1*: `targetCells[cell] = targetCells.get(cell,0) + 1`.
One last thing remaining is to sort the targetCells entries by the scores of the cells, i.e., by values, and to pick the cell with the top score. In Python one can sort dictionary entries by value using the `sorted` function from the `operator` library. NB! we need to import the library: `import operator`. The line `sortedTargetCells = sorted(targetCells.items(),`

key=operator.itemgetter(1), reverse=True) returns a list of tuples (*cell, score)*, ordered by scores, in reverse order. We just need to take the first element of the list. In particular, we take the first element of the sorted list (sortedTargetCells[0]), and since it is a move-score tuple, we take the first element of that tuple (sortedTargetCells[0][0]), which as we remember is a string that follows the dictionary format of the game moves. Hence, we can ask Python to interpret it as a dictionary using the function eval() that we have already seen today when parsing the boards: move = eval(sortedTargetCells[0][0]).

We are now finished with processing the boards! Now you need to correctly incorporate the code in the calculateMove() function. Remember that we want to execute it in the hunt mode, and to perform search in the target mode. Good luck!

[slide 14]
One more thing that we have to discuss when using the boards statistics is *data sparsity*. The term is usually used in cases when the structure of the data items we are using is quite complicated, has a lot of parameters and hence can be of huge variability (just think about how many different ship arrangements can there possibly be!). In case of such data, it is very common that whenever we have a finite set of data items, it is quite likely that the dataset will not contain a specific data item we are looking for.
Let us look at the problem of data sparsity in the Battleships game. In the dataset we have 60,000+ boards which is actually not that many, and in fact there will be cases when no similar boards can be found for a specific state of the opponent's board. Let us consider two examples. In the first board, we are in one of the first rounds of the game, so far we have managed to open only one cell on the opponent's board, and all that is required for a board to be a similar board is that it does not contain a ship in the *7*th cell of the *2*nd row. Naturally there are hundreds of such boards in our dataset, *49,429* to be more precise.
However, if we are in one of the last rounds of the game, and almost all cells on the opponent's board are open, the description of a similar board becomes too specific, and there are little chances we can find anything at all. In fact, for the board from example *2* there are no similar boards in our dataset. What can we do about it?

[slide 15]
Well, there are two strategies that you can easily implement: if no similar boards are retrieved, you either (a) use the good old deployRandomly() function and choose a random move, or (b) relax the conditions for similar boards, performing *soft comparison* when a board is similar to the opponent's board if it agrees not on all, but on some opened cells.
It is essential that you implement one of the strategies for handling data sparsity, otherwise you will receive an *Out of bounds* error at the later rounds of any game when the similarBoards list will be empty.

[slide 16]
So, to recap, we have implemented two strategies of the Battleships game: whenever we are in the hunt mode, looking for ships, we rely on hints from previous data and choose cells that are more likely to contain ships; when we are in the target mode, having hit a part of a ship, we use exhaustive search to destroy the rest of the ship. Have these strategies improved the bot performance?

[slide 17]
It can be easily checked by running the old (default) and the new versions of the bot against the same opponent, e.g., *housebot-practice*, a certain number of times and to compare the number of wins in both cases. A hint: when a bot is launched, one can tick a box *"Play another game when complete"*, and the next game will start automatically.