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.

[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: `boards = 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")`). The resulting object is a list named `boards`. You can 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*.

[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.*
In order to find similar boards, we need to …

[slide 11]

[slide 12]

[slide 13]

[slide 14]

[slide 15]

[slide 16]