ROB311: Artificial Intelligence

Project #3: Decision Trees and Adversarial Games

Winter 2021

Overview

In this project, you will expand your understanding of two important lecture topics: decision tree learning and the design of game-playing agents. The goals are to:

- implement and understand basic decision tree learning, using the greedy information gain and gain ratio splitting criteria described in the lectures; and
- build a simple game-playing agent (to be put to the test against your classmates' agents!)

The project has two parts, worth a total of **50 points**. All submissions will be via Autolab; you may submit as many times as you wish until the deadline. To complete the project, you will need to review some material that goes beyond that discussed in the lectures—more details are provided below. The due date for project submission is **Monday, March 29, 2021, by 23:59 EDT**.

As in Project #2, each part already has some basic code in place for you to start with. For example, in `decision_trees.py`, you are shown how your decision tree learning algorithm will be called by the Autolab grader.

Part 1: Decision Tree Learning

Decision trees are useful for a wide range of machine learning tasks, and have the advantage of being readily understandable. We will be concerned with building trees using training data involving discrete-valued attributes. You will implement functions to determine how to split the data (and build the tree) based on the information gain and gain ratio criteria. Your tasks are to:

1. Implement five basic support functions for decision tree learning that compute (in order): discrete entropy, conditional entropy, intrinsic information, information gain, and information gain ratio. Use the function templates `decision_tree.py` in the handout code to get started.
2. Implement a Python version of the Decision Tree Learning algorithm given on pg. 702 of the AIMA text, in the file `decision_tree.py`. Your learning implementation will make use of the functions provided above (and enable splitting using both information gain and gain ratio criteria).

Some utility code has been provided for you in the form of a `TreeNode` class and its methods, and a function to compute the 'plurality value' of a set of examples — do not modify these. Please read all comments in the starter code for implementation details. You will submit your implementations of all 6 template functions in `decision_tree.py` through Autolab.

Part 2: Adversarial Games

One of the fist applications of AI was to games, and many well known games have now been solved (e.g., you can read the fascinating story about solving checkers here). That is, the optimal set of moves given any starting state is known. However, developing game-playing agents can still be challenging! In this open-ended portion of the project, you will write an agent to play a modified version of rock-paper-scissors with a game (payoff) matrix of the form:

$$\mathbf{M} = \begin{bmatrix} 0 & -a & b \\ a & 0 & -c \\ -b & c & 0 \end{bmatrix}, \tag{1}$$

where a, b, and c are all positive (you will not be given their values). Your task is to:

1. Write a game-playing agent that attempts to win as many games as possible. The class `StudentAgent` in `iterated_single_move_games.py` has three methods that you must implement. As usual, you may only use the import statements present in the file.

Your agent will be pitted against other agents: for each given opponent, your `StudentAgent` class will repeat 1000 rounds against that opponent. See the `play_game` function in `iterated_single_move_games.py` for details. The other agents you are up against are:

- a dumb agent that always chooses the first move (see `FirstMovePlayer`),
- a copycat agent that always copies its opponent's last move (see `CopycatPlayer`),
- an agent that randomly chooses one of the three options with equal probability (see `UniformPlayer`),
- a 'goldfish' agent with very short memory (source code is not available to you),
- an agent that uses a mixed Nash equilibrium strategy (source code is not available to you),
- and an agent that follows a random Markov process that depends on the last round (source code is not available to you).

You do not have to beat every agent head-to-head, but you must win the lion's share of the points (see the Grading section below). The exact strategy that your agent employs is a design decision—however, you must briefly document the technique(s) you used in your code. As the final part of the project, we will hold a round-robin tournament. A portion of your grade on this part of the project will depend on your algorithm's performance in the tournament. Grading details are provided in the section below. Please read all the comments in the starter code for implementation details. You will submit your implementation of `StudentAgent` in `iterated_single_move_games.py` through Autolab.

Grading

We would like to reiterate that **only** functions and methods implemented for the Autolab's grader will affect your marks. As usual, submit your code to Autolab as follows:

```
tar cvf handin.tar decision_tree.py iterated_single_move_games.py
```

Points for each portion of the project will be assigned as follows:

- Decision Tree Learning – **25 points** (5 tests $\times$ 3 points per test; 2 tests $\times$ 5 points per test)

The first five tests are of the utility functions for decision tree learning (defined in Part 1); each test is designed to ensure that your support function is operating correctly. The final two tests will involve learning a tree using a hold-out (hidden) dataset.

- Adversarial Games – **25 points** (5 points for tests; 5 points for algorithm description; 15 points for tournament performance)

  Tests: The test will pit your agent in a tournament against the simple agents mentioned above. Note that each agent will play 1000 games against each opponent, and the total run time must not exceed 10 seconds. You must also score over 6500 points to get full marks, or over 3250 points to get half marks.

  Algorithm Description: In addition to your code, *you must also include a longer comment block* (approximately 10-15 lines, 80 characters per line) that describes your approach and/or the algorithm that you implemented. This comment is to be placed in the docstring right below the header of `StudentAgent`.

  Tournament: After the project submission deadline, we will run all agents head-to-head in a round robin tournament. The class's algorithms will be ranked and split into tertiles. The top scoring tertile gets 15 points, the middle tertile gets 10 points, and the bottom scoring tertile gets 5 points.

Total: **50 points**

Grading criteria include: correctness and succinctness of the implementation of support functions, proper overall program operation, and code commenting. Please note that we will test your code *and it must run successfully*. Code that is not properly commented or that looks like 'spaghetti' may result in an overall deduction of up to 10%.